Joint Post-proceedings of the First and Second International Conference on Microservices

Microservices 2017, October 2017, University of Southern Denmark, Odense, Denmark
Microservices 2019, February 2019, University of Applied Sciences and Arts Dortmund, Dortmund, Germany

Edited by

Luís Cruz-Filipe
Saverio Giallorenzo
Fabrizio Montesi
Marco Peressotti
Florian Rademacher
Sabine Sachweh

OASICS

OASIcs – Vol. 78 – Microservices 2017/2019   www.dagstuhl.de/oasics

Editors

Luís Cruz-Filipe
University of Southern Denmark, Denmark
lcf@imada.sdu.dk

Saverio Giallorenzo
University of Southern Denmark, Denmark
saverio@imada.sdu.dk

Fabrizio Montesi
University of Southern Denmark, Denmark
fmontesi@imada.sdu.dk

Marco Peressotti
University of Southern Denmark, Denmark
Peressotti@imada.sdu.dk

Florian Rademacher
University of Applied Sciences and Arts Dortmund, Germany
florian.rademacher@fh-dortmund.de

Sabine Sachweh
University of Applied Sciences and Arts Dortmund, Germany
sabine.sachweh@fh-dortmund.de

ACM Classification 2012
Applied computing → Service-oriented architectures; Software and its engineering → Software architectures; Information systems → RESTful web services; Software and its engineering → Cloud computing; Software and its engineering → Organizing principles for web applications; Software and its engineering → Development frameworks and environments

ISBN 978-3-95977-137-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-137-5.

Publication date
February 2020

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.Microservices.2017-2019.0
ISBN 978-3-95977-137-5 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Regular Papers

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

About the Microservices Community and the Microservices Conference Series. The Microservices Community[1] is one of the largest non-profit organisations purposed at sharing the knowledge and fostering collaborations on microservices. The organisation counts a broad composition of members from research institutions, private companies, universities, and public organisations.

The International Conference on Microservices (shortened, Microservices) is a conference series aimed at bring together industry and academia, to foster discussion on the practice and research of all aspects of microservices: their design, programming, and operations. Microservices is the flagship conference among the many dissemination events supported by the Microservices Community.

Microservices 2017, October 2017, Odense, Denmark. The First International Conference on Microservices[2] took place in Odense, Denmark from October 25–26, 2017, and was organised in collaboration between the University of Southern Denmark and the University of Bologna.

The program committee (PC) consisted of 16 members and the conference received the submission of 18 extended abstracts, of which 16 were accepted for presentation. The program also featured invited keynotes by Steve Ross-Talbot and Claudio Guidi. The conference counted 40+ participants, of which 58% were from industry (e.g., Red Hat Inc., Yoti Ltd., etc.) and 42% from academia.

Microservices 2019, February 2019, Dortmund, Germany. The Second International Conference on Microservices[3] took place in Dortmund, Germany from February 19–21, 2019. It was organised in collaboration between the Dortmund University of Applied Sciences and Arts and the University of Southern Denmark.

Microservices 2019 built upon the success of the previous edition. The PC consisted of 26 members and the conference received the submission of 32 extended abstracts, of which 28 where accepted for presentation. The contributions covered a broad spectrum of topics related to the three main themes around Microservices: design, development, and deployment/operations. Thus, each of the three conference days could be dedicated to one of those themes. The program featured invited keynotes by Olaf Zimmermann, Fabrizio Montesi, Ramón Medrano Llamas, Jörn Esdohr, and Peter Rossbach. The conference counted 70+ participants, of which 36% were from industry (e.g., Google LLC, Siemens AG, NGINX Inc., IBM Inc., etc.) and 64% from academia.

Post-proceedings of Microservices 2017/2019. The present volume compiles contributions from attendees of Microservices 2017 and 2019. The volume received 9 submissions of which 5 were accepted for inclusion after peer review and rebuttal.

In addition to the contributed papers, this volume includes the abstracts of the keynotes presented at the first two editions of Microservices.

[1] `https://www.microservices.community`
[2] `https://www.conf-micro.services/2017`
[3] `https://www.conf-micro.services/2019`

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We thank the authors of all submitted proposals for their work in preparing and presenting their contributions. We hope that they found the feedback from the reviewing process helpful. We also thank the members of the program committees of Microservices 2017, Microservices 2019, and these post-proceedings for their excellent work and enthusiasm.

Finally, we want to thank all the donors that provided financial support to the conference and these proceedings: Adesso AG, InnoQ GmbH, Materna SE, italianaSoftware S.r.l, Vanderlande BV, the Department of Mathematics and Computer Science of the University of Southern Denmark, and the Institute for the Digital Transformation of Application and Living Domains of the University of Applied Sciences and Arts Dortmund.

Let Industry and Academia meet.

Luís Cruz-Filipe
Saverio Giallorenzo
Fabrizio Montesi
Marco Peressotti
Florian Rademacher
Sabine Sachweh

Microservices 2017

Microservices 2017 Organisation

Program Chairs	Luís Cruz-Filipe (University of Southern Denmark)
	Fabrizio Montesi (University of Southern Denmark)
Local Organisers	Barbara Tvede Andersen (University of Southern Denmark)
	Ronald Jabangwe (University of Southern Denmark)
	Marco Peressotti (University of Southern Denmark)
Publicity Chair	Saverio Giallorenzo (University of Bologna / INRIA)
Program Committee	Farhad Arbab (Leiden University and CWI)
	Pierre-Malo Deniélou (Google)
	Nicola Dragoni (Technical Univ. of Denmark / Örebro University)
	Schahram Dustdar (TU Wien)
	Saverio Giallorenzo (University of Bologna / INRIA)
	Claudio Guidi (italianaSoftware)
	Alberto Lluch Lafuente (Technical University of Denmark)
	Einar Broch Johnsen (University of Oslo)
	Sung-Shik Jongmans (Open University of the Netherlands)
	Manuel Mazzara (Innopolis University)
	Kevin Ottens (KDAB)
	Steve Ross-Talbot (Estafet)
	Gwen Salaün (Inria Grenoble - Rhône-Alpes)
	Nobuko Yoshida (Imperial College London)
	Ingrid Chieh Yu (University of Oslo)
	Olaf Zimmermann (Univ. of Appl. Sciences of Eastern Switzerland)

Microservices 2017 Keynote Abstracts

A Linguistic Approach to Microservices, Claudio Guidi, italianaSoftware S.r.l.

Microservices are usually considered the be technology agnostic, thus they are approached in terms of architectures or models to be applied on distributed systems. Nevertheless, their basic mechanisms can be crystallized within a unique programming language by offering a new mindset for developers and engineers, In the past years we dealt with such an objective starting from the theoretical foundations of service oriented computing. In this presentation I'll show our experience and our results in approaching microservices with a specific programming language called Jolie.

The problem with Microservices, Steve Ross-Talbot, Estafet

Are microservices really the next Big Thing? Whilst they currently dominate conferences and the language of product vendors — often linked to APIs, Continuous Delivery, Containers and PaaS — the deluge of information often confuses rather than clarifies. This has led to sub-optimal microservices architectures and some spectacular failures.

In this talk, we will discuss both data-centric and interaction-centric approaches, looking at what happens when people "code-first-and-ask-questions-later". How much up-front thinking do you need?, and how can you exert sufficient control over implementations once

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

they are live? The future may well include policy-driven microservices architecture, but you need to ask some tough questions now if you are going to deliver to the business on a continued and scaled basis.

Microservices 2019

Microservices 2019 Organisation

Program Chairs	Saverio Giallorenzo (University of Southern Denmark)
	Marco Peressotti (University of Southern Denmark)
	Florian Rademacher (Univ. of Appl. Sciences and Arts Dortmund)
	Sabine Sachweh (Univ. of Applied Sciences and Arts Dortmund)
Local Organisers	Philipp Heisig (University of Applied Sciences and Arts Dortmund)
	Philip Wizenty (Univ. of Applied Sciences and Arts Dortmund)
Publicity Chairs	Barbara Tvede Andersen (University of Southern Denmark)
	Jonas Sorgalla (University of Applied Sciences and Arts Dortmund)
Program Committee	Farhad Arbab (Leiden University and CWI)
	Luís Cruz-Filipe (University of Southern Denmark)
	Pierre-Malo Deniélou (Google)
	Claudio Guidi (italianaSoftware)
	Marcel Hahn (University of Kassel)
	Philipp Heisig (University of Applied Sciences and Arts Dortmund)
	Thomas Hildebrandt (University of Copenhagen)
	Pooyan Jamshidi (University of South Carolina)
	Sung-Shik Jongmans (Open University of the Netherlands)
	Michalis Kargakis (Red Hat)
	Ivan Lanese (University of Bologna)
	Sanja Lazarova-Molnar (Maersk Mc-Kinney Moller Institute)
	Fei Li (Siemens)
	Ramón Medrano Llamas (Google)
	Jacopo Mauro (University of Southern Denmark)
	Martin Peters (com2m GmbH)
	Marco Prandini (University of Bologna)
	Steve Ross-Talbot (Estafet)
	Alessandro Rossini (PwC Consulting)
	Larisa Safina (Innopolis University)
	Gwen Salaün (Inria Grenoble, Rhône-Alpes)
	Jonas Sorgalla (Univ. of Applied Sciences and Arts Dortmund)
	Balakrishna Subramoney (SunBio IT Solutions)
	Stefan Tilkov (innoQ Deutschland GmbH)
	Olaf Zimmermann (Univ. of Appl. Sciences of Eastern Switzerland)
	Albert Zündorf (University of Kassel)
Additional reviewers	Mirco Lammert (Univ. of Applied Sciences and Arts Dortmund)
	Andreas Püsche (Univ. of Applied Sciences and Arts Dortmund)
	Philip Wizenty (Univ. of Applied Sciences and Arts Dortmund)
	Stefano Pio Zingaro (University of Bologna)

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Microservices 2019 Keynote Abstracts

Domain-Specific Service Decomposition with Microservices API Patterns, Olaf Zimmermann

Service orientation is a key enabler for cloud-native application development. Microservices have emerged as a state-of-the-art implementation approach for the realization of the Service-Oriented Architecture (SOA) style, promoting modern software engineering and deployment practices such as containerization, continuous delivery, and DevOps. Designing (micro-)services interfaces' to be expressive, responsive, and evolvable is challenging. For instance, deciding for suited granularities is a complex task resolving many conflicting forces; one size does not fit all. Domain-Driven Design (DDD) can be applied to identify and specify service boundaries. However, service designers seek concrete, actionable guidance going beyond high-level advice such as "turn each bounded context into a microservice". Interface signatures and message representations need particular attention as their structures influence the service quality characteristics. This presentation first recapitulates prevalent SOA principles, microservices tenets and DDD patterns. It then reports on the ongoing compilation of complementary Microservices API Patterns (MAP) and proposes a set of pattern-based API refactorings for service decomposition. Finally, the presentation highlights some of the related research and development challenges.

Understandable Microservices, Fabrizio Montesi

Microservices come at a price: they have to be integrated to get a meaningful application. This motivated the creation of tools that make integration easier. Today, we spend more time developing integration than actual applications, so this development could not have come at a better time. Enter Jolie, a microservice-oriented programming language. By offering native linguistic features for composing microservices, Jolie has become a swiss army knife that can be used by integration ninjas and wise software designers that plan for maintanable software. To explore how microservices and Jolie can make us productive with integration, we'll develop microservices for a concrete business idea: a publishing platform for sharing Chuck Norris jokes. Technically, we'll create an API gateway for two different third-party Internet websites, integrate their behaviours, and ultimately get what we want without having to host either of them. Would you write object-oriented software without an object-oriented language? Ask yourself again, but for microservices, after you see this talk.

Engineering Reliability, Ramón Medrano Llamas

How do you scale up a service, so it can serve millions (or billions!) of users around the globe, make it reliable and fast while maintaining development speed and change safety? This talk introduces Site Reliability Engineering (SRE) at Google, explaining its purpose and describing the techniques it uses and the challenges it addresses. SRE teams manage Google's many services and properties, plus all the brand new Cloud infrastructure from our offices worldwide. They draw upon Linux based computing resources that are distributed in several data centres around the globe to deploy, manage, and serve globally available services four billions of users.

Factory of Things - Using Microservices for Data Processing and IoT, Jörn Esdohr

Industrial devices and machines, ranging from lights over elevators to complete factories, are producing large amounts of data. Some of it is used to grant simple ad-hoc monitoring and control capabilities, but a lot of this valuable resource ends up discarded due to the lack of a comprehensive data processing infrastructure. Microservices provide a reliable and performant architecture for the Internet of Things (IoT) to connect devices on a large scale, which provides a path to collect and analyse the flood of accumulating application data. At com2m, a containerized microservice-based IoT platform was developed leveraging graph and document databases, and modern web technologies. We present the Factory of Things as a showcase that demonstrates the real-world integration of a manufacturing line powered by programmable logic controllers. The IoT platform enables new data processing possibilities to monitor devices and enables the development of rich data-based services.

Build Fashionable Container Systems with Microservices, Clouds, and Kubernetes, Peter Rossbach

Transform your organization and systems so that they no longer need an end-state. Modern clouds and the container technology help you to build self-healing autonomous scalable systems around the globe. The Cloud Native Computing Foundation ecosystem offers you many features to setup and manage complex cloud-native container systems. Serverless or microservice architectures need a lot of glue infrastructure components. In this talk I will show you some automation practices, such as infrastructure as code, release automation, and container orchestration. We build container systems in conjunction with Kubernetes and Clouds. As a developer you will learn how you can easily control your stage environments, reuse setups, and how to release your complete application stack with cloud-native technologies.

Scientific Committee of the post-proceedings

Einar Broch Johnsen (University of Oslo)
Claudio Guidi (italianaSoftware)
Philipp Heisig (University of Applied Sciences and Arts Dortmund)
Sung-Shik Jongmans (Open University of the Netherlands)
Ivan Lanese (University of Bologna)
Fei Li (Siemens)
Jacopo Mauro (University of Southern Denmark)
Manuel Mazzara (Innopolis University)
Marco Prandini (University of Bologna)
Larisa Safina (Innopolis University)
Gwen Salaün (Inria Grenoble, Rhône-Alpes)
Jonas Sorgalla (University of Applied Sciences and Arts Dortmund)
Stefan Tilkov (innoQ Deutschland GmbH)
Olaf Zimmermann (University of Applied Sciences of Eastern Switzerland)
Stefano Pio Zingaro (University of Bologna)
Albert Zündorf (University of Kassel)

Using Microservices to Customize Multi-Tenant SaaS: From Intrusive to Non-Intrusive

Hui Song
SINTEF, Oslo, Norway
hui.song@sintef.no

Phu H. Nguyen[1]
SINTEF, Oslo, Norway
phu.nguyen@sintef.no

Franck Chauvel
SINTEF, Oslo, Norway
franck.chauvel@sintef.no

Abstract

Customization is a widely adopted practice on enterprise software applications such as Enterprise resource planning (ERP) or Customer relation management (CRM). Software vendors deploy their enterprise software product on the premises of a customer, which is then often customized for different specific needs of the customer. When enterprise applications are moving to the cloud as mutli-tenant Software-as-a-Service (SaaS), the traditional way of on-premises customization faces new challenges because a customer no longer has an exclusive control to the application. To empower businesses with specific requirements on top of the shared standard SaaS, vendors need a novel approach to support the customization on the multi-tenant SaaS. In this paper, we summarize our two approaches for customizing multi-tenant SaaS using microservices: intrusive and non-intrusive. The paper clarifies the key concepts related to the problem of multi-tenant customization, and describes a design with a reference architecture and high-level principles. We also discuss the key technical challenges and the feasible solutions to implement this architecture. Our microservice-based customization solution is promising to meet the general customization requirements, and achieves a balance between isolation, assimilation and economy of scale.

2012 ACM Subject Classification Software and its engineering → Software as a service orchestration systems; Software and its engineering → Cloud computing; Applied computing → Service-oriented architectures

Keywords and phrases Customization, Software-as-a-Service (SaaS), Microservices, Multi-tenancy, Cloud, Reference Architecture

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.1

Funding This work is funded by the Research Council of Norway under the grant agreement number 256594 (the Cirrus project).

Acknowledgements We want to thank our colleagues at Supper Office and Visma for the fruitful collaboration in the Cirrus project.

1 Introduction

Most companies rely on enterprise software applications to drive their daily business, such as Enterprise resource planning (ERP) or Customer relation management (CRM). Because every company is unique, a standard product application cannot fit all the requirements of any company, and therefore often needs to be customized for individual customers. In

[1] Corresponding author

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 1; pp. 1:1–1:18
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

practice, a customer can easily spend ten times the cost on customization than the licence they bought for the original application [3]. Software customization is traditionally done by re-defining work flows, developing add-in applications, or even directly modifying the source code of the standard product application. In this paper, we differentiate *customization* from *configuration*: The former involves software development work whereas the latter only involves changing some values of the predefined parameters. Advanced customers often require features that are not predictable for vendors, making customization inevitable, since configuration is limited within the features that are already implemented by the vendors.

Following the trend of cloud computing, enterprise software vendors are moving from single-tenant on-premises applications to multi-tenant (Cloud-based) Software as a Service (SaaS) [6]. Customer companies no longer buy a license from the vendor and install it in their own premises. Instead, they subscribe to an online service, which is also used by other customers, known as *tenants* of the service. The SaaS model brings new challenges to the software vendors with regard to enabling customization. It is not possible for any tenant to directly edit the source code of the same product service shared by other tenants. Software vendors must enhance the SaaS model with the ability to enable tenant-specific customization in the multi-tenant context. Under such setup, customization on multi-tenant SaaS must meet three basic requirements. These requirements have been defined together with the two software vendors who are the industrial partners in the Cirrus project.

- *Isolation*: The customization for one tenant must not affect the other tenants. Tenant isolation, especially in terms of security is of paramount importance.
- *Assimilation*: Customization should not harm the performance and user experience of the SaaS. In other words, the “look and feel” of the SaaS with customization should not change compared to the original SaaS.
- *Economy of scale*: With more customers subscribe to customize the SaaS, the average cost per customer should decrease. The SaaS business model allows to make full use of the economy of scale, as multiple tenants (customers) share the same application and database instance [1]. Enabling customization for multi-tenant SaaS should still ensure the economy of scale brought by the SaaS business model.

The state of the art and practice on enabling customization for multi-tenant SaaS may still be at an early stage discussed as follows. There are enterprise software vendors that move their products to SaaS without supporting the same level of customization capabilities as their customers used to have on their own premises. As a result, a significant number of their customers are not following to the cloud [7]. Without customization capabilities, the customers lost an important weapon for tailoring the services according to their real requirements, and for continuous business innovation. Some vendors choose to support either lightweight, in-product customization by providing customers with scripting or work flow languages [24]. In this way, the customization capability is stronger than parameter configuration but still limited by the languages. It is also not ideal in terms of isolation, as the scripts are running inside the main product. Other vendors, especially the big ones such as Salesforce, choose a heavyweight direction, transforming themselves from a product into a development platform for customers to implement their own applications [21]. In this way, the customization in terms of standalone applications does not meet the assimilation requirements, as the external applications will break the consistent user experience of the product service, and also drag down the response time. More importantly, this solution requires huge investment from vendors and strong expertise from customization developers.

Using microservices is a promising direction to customize multi-tenant SaaS because microservices architectures offer several benefits. First, microservices for customization purposes can be packaged and deployed in isolation from the main product, which is an important requirement for multi-tenant context. Moreover, independent development and deployment of microservices ease the adoption of continuous integration and delivery, and reduce, in turn, the time to market for each service. Independence also allows engineers to choose the technology that best suits one and only one service, while other services may use different programming languages, database, etc. Each service can also be operated independently, including upgrades, scaling, etc. In this paper, we discuss our two approaches of using microservices to enable customization for multi-tenant SaaS: intrusive and non-intrusive. The intrusive approach [22, 2] prescribes that the main body of customization code runs in a separate microservice, isolated from the main service (of the main product), whilst specific parts of the customization code are sent back to the main product and dynamically compiled and executed within the execution context of the main service. While intrusive customization using microservices is technically sound, its practical adoption by industry may be hindered by the intrusive way of customization code, which would be developed by "third-parties" that cannot be trusted by the software vendor to be dynamically compiled and executed within the execution context of the main service. Thus, we have evolved our approach to become non-intrusive [14, 15]. The non-intrusive approach avoids using intrusive call-back code for customization and rather orchestrates customization using the API Gateway pattern [19]. Via API Gateway(s), the APIs of the main product and the APIs of the microservices implementing customization are exposed for tenant-specific authorized access. We have demonstrated the two proposed approaches by two experimental use cases of transforming two Microsoft's reference .Net Core web applications into customizable SaaS: MusicStore [12] and eShopOnContainers [11].

Based on these two approaches, we generalize our approaches by providing a reference architecture of customizing multi-tenant SaaS by microservices, together with the general principles to achieve the requirements of isolation, assimilation and economy of scale. Whether intrusive or non-intrusive, our work has provided the designs and experiments towards novel, cloud-native architectures for customizing multi-tenant SaaS by tenant-specific microservices. A customization for a particular tenant is running as a standalone service and dynamically registered to the product service for this tenant. At runtime, the customization microservices are triggered by the product service when the latter reaches a registered extension point. They communicate with each other via REST APIs. The customization microservices are hosted by the same vendor cloud as the product service.

This paper is a report based on the investigation, design and experiments under the collaboration among a research institute and two software vendors. The objective of this paper is to provide: 1) a sample solution in the direction of using microservices in a high abstraction level, aiming at inspiring other vendors having the same customization problem for multi-tenant SaaS; and 2) a reference for researchers interested in the problem of multi-tenant customization, with a clarification of relevant concepts and research challenges. The contributions of this paper can be summarized as follows.

- We clarify the problem of muti-tenant SaaS customization with a conceptual architecture, which defines the high-level concepts involved in the problem and the relationships between these concepts.
- We provide a reference architecture of customizing multi-tenant SaaS by microservices, together with the general principles to achieve the requirements of isolation, assimilation and economy of scale.
- We identify a set of technical challenges towards implementing this reference architecture, and propose technical solutions towards these challenges.

The remainder of this paper is organized as follows: In Section 2, we give a motivational example to demonstrate the challenges of deep customization. Then, Section 3 provides a conceptual architecture of multi-tenant SaaS. Sections 4 and 5 describe our intrusive and non-intrusive approaches for enabling the customization of multi-tenant SaaS using microservices.

We generalize our solutions and provide a reference architecture for customization using microservices in Section 6. After that, Section 7 discusses the technical details in the proposed reference architecture. Section 8 presents the related work. Finally, we give our conclusions and future work in Section 9.

2 A Motivational Example

Let us consider *MuTeShop.com* (Multi-Tenant Shops) as a made-up example that captures the requirement of customization. *MuTeShop.com* offers web-based online shopping SaaS: Customers can quickly set up their own shopping website. From the *MuTeShop.com* software vendor's point of view, each customer is a *tenant* with a separate website for their *end-users* to browse and buy goods.

MuTeShop.com has to be customizable. For example, one of their key customer/tenant, e.g., *Music.MuTeShop.com*, requires that their shopping cart includes a charity donation option. Whenever an end-user adds an album into her shopping cart, she can donate some money to a designated charity, which eventually adds-up on the total checkout price. *Music.MuTeShop.com* hires a third-party consultant to implement this customization, which involves the following changes to the standard *MuTeShop.com* product. They need to change: *(i)* the database storage to be able to record the amount of donation for each item in the shopping cart, *(ii)* the business logic to account for these donations, and *(iii)* finally the user interface for end-users to choose/see how much they donate.

As a multi-tenant SaaS, *MuTeShop.com* cannot allow the consultant to modify their product source code to implement such customization, because the same database schema and the price accounting source code are shared by multiple tenants. Modifying the code for one tenant would interfere with the service to other tenants. Instead, the product service of *MuTeShop.com* should only provide the standard features that are common for all the tenants. The customization required specifically by *Music.MuTeShop.com* should be running in an isolated way, outside of the product service. When registered to the product service, this customization should modify the behaviour of the product service as stated above if and only if the user requests are bound to this tenant.

The example illustrates the problem that many SaaS vendors face: Their services are successful for some customers only if they can do deep customization. But the vendor cannot allow the same way of deep customization as for on-premises products, because they have to keep the service multi-tenant. In summary, they need a customization solution that achieves both *isolation* and *assimilation*.

3 A Conceptual Model of Customization for Multi-Tenant SaaS

Figure 1 summarizes the main concepts related to the customization of multi-tenant SaaS. There are three different roles in the ecosystem of multi-tenant SaaS customization, i.e., the Vendors who provide the main SaaS, the Customers who subscribe to the service, and the Consultants who are hired by the customers to customize the SaaS. In practice, the three roles are not necessarily taken by different companies, e.g., a customer company may have their own IT team and developers that are competent for doing customization. A vendor company may also have its own consult team who sells their own service and does customization under the customer's request.

The Product is the software produced by the vendor. It may have multiple versions. A Product Service is a running instance of a specific product version, hosted by a Product Environment, together with the Product Data. A product environment is a specific Environment,

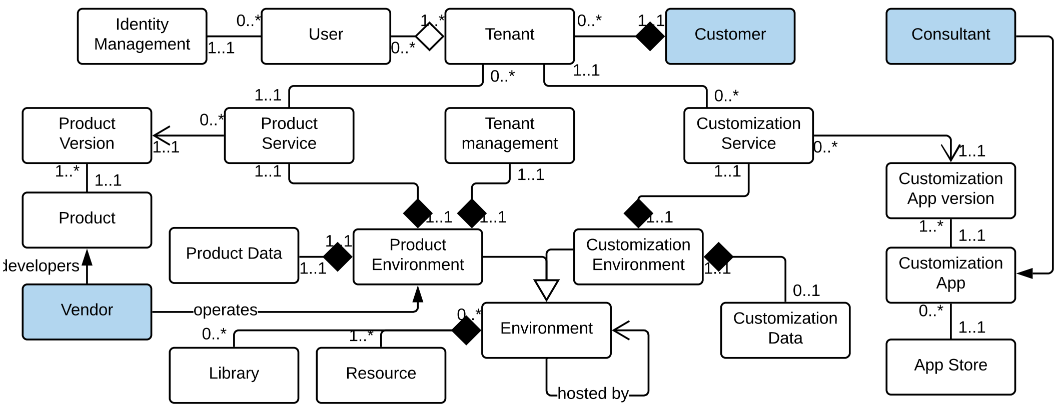


Figure 1 A conceptual model of customization for multi-tenant SaaS.

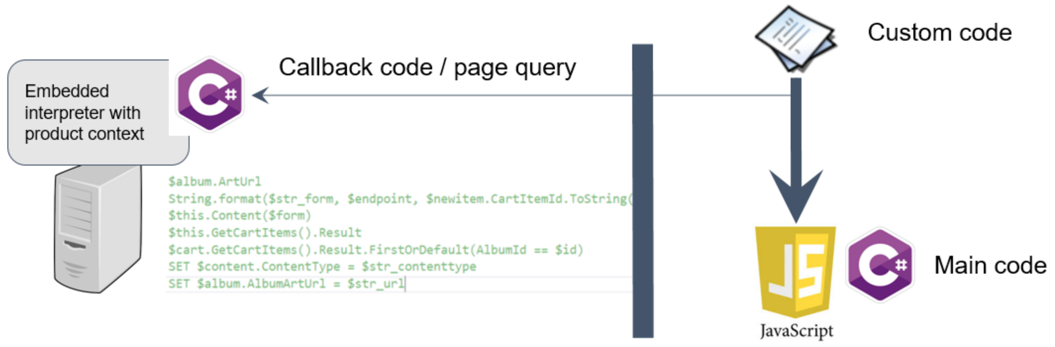
which is a self-contained computation unit with software, Libraries and Resources, such as a Docker container or a virtual machine. A product service typically runs in an exclusive environment, meaning that one environment is for one and only one instance. An environment may be hosted by another environment, e.g., a Docker container may be hosted by a virtual machine, and the latter is hosted by a cloud infrastructure.

One product service serves multiple Tenants, and at the same time, the vendor manages the product environment. A vendor usually operates multiple product services (and therefore multiple product environments) at the same time: They may need to maintain instances for several versions of the product for their customers. For the same version, they usually run one main instance for the production usage, one staging instance for user testing, and one development instance for testing and debugging. Even within the product usage, there may need several product instances due to load capacity or region constraints. Under such setup, a customer may have the subscription of multiple product instances, each of which is subscribed via a different tenant. In other words, every tenant belongs to one and only one product instance. This simplifies the billing and management for the vendors. Each tenant covers a number of Users, which are the persons who have the right to access the product instance via this tenant. They are typically employees of the customer behind this tenant, but a tenant may also include users from the consult company who help the customer through training, maintenance or customization.

A Customization App is software code that implements customization to the product. It may have several versions. A Customization Service is a running instance of one version of the Customization App. A customization service is hosted by a Customization Environment. Similar to a product environment, a customization environment also provides necessary resources, libraries and database to run the customization service. Database is optional as some lightweight customization services can be stateless. A customization service is registered to a tenant, and it only changes the behaviour of the product for this particular tenant.

4 Intrusive Customization Using Microservices

In our previous work, we have experimented with an approach to using (semi-)intrusive microservices for the customization of multi-tenant SaaS [22, 2]. We allow the tenants to replace the fine-grained structures, i.e., any part of user interface (UI), business logic (BL), or database (DB) in the original product by external microservices. The main customization


Custom code
Callback code / page query
Embedded interpreter with product context
JavaScript
Main code

```
$album.ArtUrl
String.format($str_form, $endpoint, $newitem.CartItemId.ToString(
$this.Content($form)
$this.GetCartItems().Result
$cart.GetCartItems().Result.FirstOrDefault(AlbumId == $id)
SET $content.ContentType = $str_contenttype
SET $album.AlbumArtUrl = $str_url
```

Figure 2 Intrusive customization code.

logic is running in those microservices, as parallel stacks outside of the main product. However, when a customization logic needs to access the product data or to manipulate the state of the product, it sends the so-called “call-back code” to the product, and the latter will interpret the call-back code dynamically during runtime. Since the call-back code is running under the same context as the replaced main product code, in theory it has the equivalent power as the main product code, which means that it can access any data and change any the states that are reachable by the product code. Therefore, this achieves the *deep customization* because what can be customized is not limited by the APIs of the main product. This way does not require the main product to provide any dedicated APIs for customization purpose.

A proof-of-concept implementation on multi-tenant customization based on intrusive microservices was conducted to an open source online shopping product, the Microsoft MusicStore [12]. The Microsoft MusicStore can be considered as an implementation of the MuTeShop.com example in Section 2. We first transformed it into a customizable mutli-tenant SaaS (Section 4.1). Then, we tested its customization capability by programming a simple microservice realizing the customization scenario as described in Section 2. The following proof-of-concept implementation described in Section 4.2 demonstrates the usage of synchronous triggering, intrusive invocation, separate NoSQL database, and Docker-based environments. The experiment in Section 4.2 shows that, within a reasonable cost, it is possible to enable microservice-based customization on originally un-customizable product.

4.1 Adapting the SaaS to be Customizable

We adapted the source code of MusicStore to enable microservice-based customization by adding a generic library and perform a code rewriting. A simple *tenant manager* is used to register the mapping from original methods of the main code to customization code. Figure 2 shows how the customization code interacts with the main code flow of the main product. A generic *interceptor* drives the synchronous triggering between the main product and the customization code. The *interceptor* sends the current execution context from the main product to the corresponding customization microservice for executing customization logic. After executing customization logic, the customization microservice sends callback code to the main product to apply the customization and even request for other context from the main product if follow-up customization logic is necessary. A *callback code interpreter* executes the intrusive callback code on the local context to apply the customization in the main code.

Before building and deploying the MusicStore application as a service, we performed an automatic code rewriting to enable the triggering and callback code mechanisms. In particular, we add three pieces of code in the beginning of each method. Listing 1 below shows an example of such added code for the method *AddToCart* in the class *ShoppingCartController*. The first initializes a local context as a Dictionary object and fills it with the method parameters. This context dictionary will be used later on by the callback code interpreter. The second invokes the generic interceptor with the context and the method name. The third checks if a return value is available to decide whether to skip the original method body and return the customization results.

Listing 1 Triggering customization.

```
// first piece of code
var context = new Dictionary<string, object>()
{
    ["id"] = id,
    ["cart"] = cart,
    ["this"] = this
};
// second piece of code
ReturnValue rv = Interceptor.Intercept(
    Controllers.TenantController.currentUser,
    "MusicStore.Controllers.ShoppingCartController.AddToCart",
    false,
    context
);
// third piece of code
if(rv != null)
{
    return rv.Value;
}
```

According to our implementation practice, very light effort is required to realize the deep customization support on a legacy software application. The effort is focused on generic mechanisms, without specific consideration of the actual customization requirements or features.

4.2 Sample Customization

On the customizable MusicStore, we performed three customization use cases.

- Donation: as described in Section 2, we need to add a new page for end-users to choose the donation, a new column in the shopping cart table to show the donations and a new business logic computing total price for the shopping cart.
- Visit Counting: We want to record how many times each album has been visited. The feature needs to be triggered every time an album detail view is loaded.
- Real Cover: We want to use the album title to search the cover picture from Bing Image[2] and replace the original place-holder picture. A pop-up comment shows the picture source when the mouse cursor is hovered on the picture.

We design these use cases deliberately to achieve a good coverage of the general requirements of customization on Web-based enterprise systems. In the user interface level, they cover the changes within a web page, i.e., adding, removing or changing the position of HTML controls (UI components such as text, button, list, image, etc.), and adding a new

[2] `https://www.bing.com/images/`

page. The third use case also changed the browser-executed logics. In the business logic level, they cover the need to add or override server-side logics, override the action to particular events, change the bindings between UI controls and the data, and execute the services that are provided by a third party. In the database level, they require new tables, as well as new fields to an existing table. These requirements are summarized based on the actual customization cases on the on-premises products provided by the two companies.

We implemented the three customization scenarios in TypeScript, using the Node.js HTTP server to host the customization microservice [22, 2]. The first two scenarios request data storage, and we used MongoDB as the customer database. Node.js and MongoDB are running in two Docker containers, hosted on the same node as the product service. The entire customization code includes 384 lines of code in five TypeScript files (one file for each scenario, plus two common files to configure and launch the HTTP server) and 175 lines of new code in four Razor HTML templates (of which, two templates are new and the other two are copy-and-pasted from MusicStore, with 176 lines of code that are not changed).

The effect of the customized MusicStore can be seen by a screen-shot video[3]. In the video, we are using a MusicStore service through a fictional tenant. We first see the standard way to buy a music album through the MusicStore, i.e., browsing the album, add it to the shopping cart, and check the overview. After that, we deploy the customization code as a microservice and registered it into the MusicStore tenant manager. The effect of the customization is instant: When we repeat the process, we first see a new cover image of the album. When we add the album to shopping cart, we are led to a new page to select the donation amount, and shown a shopping cart overview with donations and a different total price. Finally, we open a new page to check the statistics about the album visits. At the end of the video, we log off the tenant, and the service immediately goes back to the standard behaviour.

The video also shows some non-functional features of the customization. First, the customization code is deployed and registered to the MusicStore at runtime, without rebooting the product service, and affects only the particular tenant. Second, the customized behaviour is seamlessly integrated into the product service: The new pages and the modified ones all keep the same UI style as the original MusicStore. From the end-user's perspective, it is not difficult to notice that the application has been customized.

The customization microservices have reasonable resource consumption, and is able to scale. A further examination shows that a customized page takes in average 100 millisecond longer to load, comparing to the original page. However, considering that the average page loading time in MusicStore is over two seconds, the slow down is tolerable. The footprint for a customization microservice under this scenario and the technical stack (i.e., Node.js, MongoDB, Docker) is minimal. The two Docker containers used 20 and 50 megabytes of memory at runtime.

5 Non-Intrusive Customization Using Microservices

Despite the ultimate assimilation, which means that the tenants are able to do anything for customization, just as if they are developers of the main product - our intrusive microservice approach for customization was finally not adopted by the software vendors who commissioned this research. The main concern is security. Since the partners are virtually capable of doing everything to the main product during run-time, it requires strict inspection by the vendors on the customization code, which is not pragmatic at the moment. As a result, we have turned to non-intrusive way for microservices-based customization, which should allow the vendors to keep the customization code of tenants under control [14, 15].

[3] https://youtu.be/IIuCeTHbcxc

To make non-intrusive customization possible, there is a main prerequisite for the web-based SaaS' architecture, i.e., the clear separation of the user interface (UI) part from the back-end business logic (BL) part [15]. This means that a web-based SaaS must be split into a WebUI part and back-end BL service(s). By separating the UI and the BL of the main product, we can introduce microservices implementing tenant-specific customization for the main product at UI, BL, and database (DB) levels. Note that we focus on web-based SaaS because of its popularity. Figure 3 shows an overview of the non-intrusive approach. Each customization for a tenant is running as a standalone microservice and dynamically registered to the main product service for this tenant. The APIs of the customization microservice are available for authorized access via API gateway. At runtime, whenever the main product service reaches a registered customization point for a tenant, the main product service triggers the corresponding customization for that tenant by calling the REST APIs of the tenant's customization microservices via the API gateway. Note that the customization microservices have been registered with the tenant manager in advance.

The WebMVC Customizer is a local component that is introduced into the main product's WebApp MVC to intercept the flows of the main product. The WebMVC Customizer is similar to the Interceptor of the intrusive approach in Section 4.1. Tenant Manager is a service that manages the customization(s) for every tenant, which is also similar to the intrusive approach. The main difference between the intrusive approach and the non-intrusive approach comes from the introduction of the API Gateways, the Identity and Access Management (IAM) service, and the Event Bus. In our non-intrusive approach, we follow the API gateway pattern [19] to decouple the client applications (e.g., the WebMVC application) from the internal microservices (for customization or main-stream BL). The key point in the non-intrusive approach is that it enables the authorized access of the tenants' customization microservices to the main product BL via the API gateways. In this way, the tenants' customization microservices can have access to the necessary execution context of the main product BL if needed and allowed. The non-intrusive approach can provide deep customization because it allows a customization service to replace a BL component of the main product for the corresponding tenant if authorized. The authorized access of the tenants' customization microservices to the main product BL components makes the deep customization manageable. This differs from the intrusive way of sending "call-back" code from customization microservices to the main product to be dynamically compiled and executed within the execution context of the main service. The IAM Service built on an OpenID Connect[4] or OAuth 2.0[5] Identity provider can make tenant-specific customization authorized. Using standardized and powerful authentication and authorization mechanisms such as OAuth 2.0 is very important for tenant-isolation at the application level, especially regarding customization. Last but not least, the Event Bus allows the customization microservices to have asynchronous event-based communication with the main product BL components for customization purposes. We have implemented a proof-of-concept of our non-intrusive approach for enabling deep customization of a reference application for microservices architecture, eShopOnContainers [11]. The MusicStore could be re-engineered to enable non-intrusive customization but we chose the eShopOnContainers because the eShopOnContainers' architecture already satisfies our prerequisites for non-intrusive customization. Due to space reason, we refer readers to [15] for more details of the proof-of-concept on eShopOnContainers.

[4] https://openid.net/connect/

[5] https://oauth.net/2/

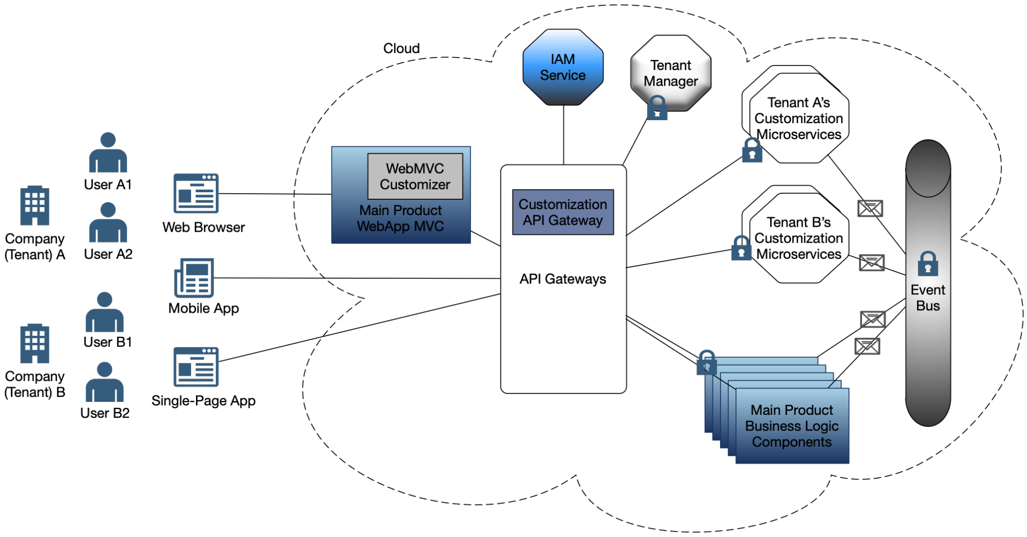


Figure 3 An overview of the non-intrusive approach [15].

6 A Reference Architecture for Customization by Microservices

This section generalizes the two customization approaches using microservices. We present the main principles for the microservice-based style for customization, and a reference architecture that follows these principles.

6.1 Principles

We adopt a microservice-based style for customization, driven by a set of high-level principles, or design decisions, in order to meet the requirements of isolation, assimilation and economy of scale. The first set of principles meets the following requirements of **isolation**.

- **Every customization is a service.** A customization should be a stand-alone running entity, with its own life-cycle independent of the product. Such a solution avoids a failure in the customization (such as dead loop) from impacting the normal operation of the main product. The interaction of customization services with the main product is monitored and administrated.
- **A customization service serves one and only one tenant.** This principle also indicates that no more than one product service connects to a same customization service, since each tenant belongs to only one product service.
- **Each customization service runs on its own environment.** This prevents customization services from influencing each other at runtime. It also simplifies the management of customization, such as monitoring and billing.
- **A customization service has its own database.** A customization should not be allowed to modify the schema of the product database. It uses its exclusive database to store the customization-specific data. A customization service does not have direct access to either the product database or the other customization database.
- **A customization service communicates with the product service and other customization services only via REST API**. Other ways of communication, such as shared database, shared memory or files, will make the services more tightly coupled.

These principles that favour isolation have negative effects on either assimilation or the economy of scale. The more isolation, the less likely that customization can be assimilated with the main product. Similarly, the more isolation leads to more use of resources, which means less favor for the economy of scale. We make the following design decisions to reduce such negative effects and bring a better balance regarding isolation, assimilation and economy of scale.

- **A customization environment is "close to" its product environment.** If a product environment is hosted by a cloud provider, such as Amazon AWS, the customization environments related to it should be deployed into an infrastructure from the same cloud provider, in the same region. This ensures the low latency of their communication.
- **The external URL to access a customization service is consistent with the product URLs.** Most of the customization services are not visible to end users, but only used indirectly when the users invoke the product. When a customization service is exposed directly to the end user, the URL to access this service should be in the same style as the product URL, so that the users do not feel the separation.
- **The vendor should provide a unified identity management for both product and customization,** so that users do not need separate login to use the customization services.
- **Customization environments should have minimal footprint**, so that the vendors can host a large scale of customers for each product service. In other words, customization services should be lightweight microservices that have minimal resource consumption.
- **The vendor should manage all the environments in an elastic way.** This will reduce the total cost of resources, especially when there is a large number of lightweight, infrequently used customization services.
- **The vendor should facilitate the reuse-by-code for customization Apps.** When a customization solution fits multiple customers (this normally happens when the customers hire the same consultant), it should be easy for the customers to reuse the customization App by code, i.e., to create a new instance of this App.

6.2 A Reference Architecture

Following the principles in the last section, we come up with a reference architecture to support the customization of multi-tenant SaaS using customer microservices. Figure 4 illustrates this reference architecture with one product instance and two customization services. The product is a typical browser-server web-based application. The single product instance serves several tenants simultaneously. The tenant management component, as part of the product instance, controls the valid tenant served by this instance, the unified identity management service controls which users have the access via each tenant. The product instance and the database are deployed in the same virtual machine from a public cloud provider. All the user requests from the browser go through a web proxy, which translate the user-friendly URL into the internal address used by the cloud provider.

Customization service customizes the behaviour of the product instance for one of its tenants, by introducing new features and replacing existing features. The customization service is registered to the tenant manager in a customization registration process before it can be triggered. The new features can be accessed via a specific URL directly from the users, or *triggered* by the product instance under predefined circumstances. The replacing features are triggered by the product instance, when the original feature in the product is about to be activated. The tenant manager defines when to trigger the customization services,

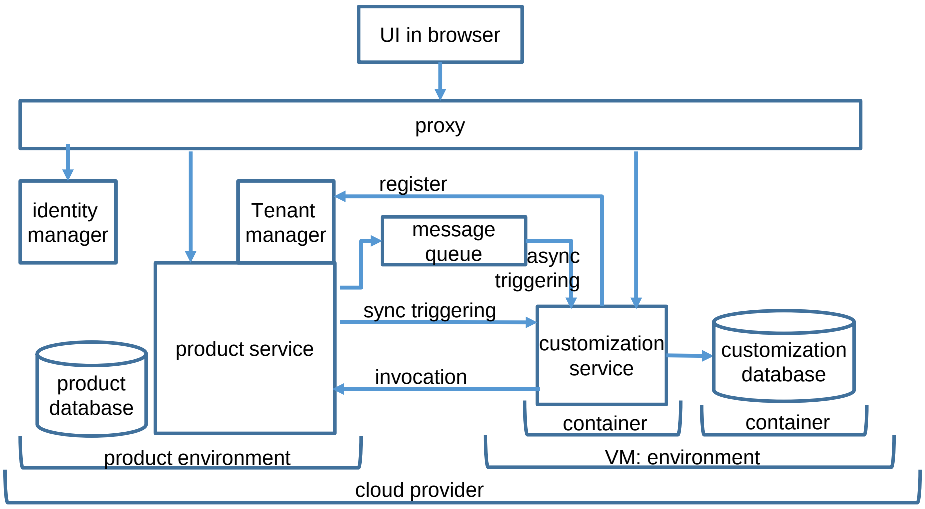


Figure 4 A reference architecture of using microservices for customization.

based on the customization registration. The customization service may need the standard data from the product instance, or modify the state and data of the product instance. It achieves this by *invoking* the product instance via API calls (in a non-intrusive approach) or call-back code (in an intrusive approach). Triggering and invoking are the two directions of communications between the product instance and the customization service. We will discuss later in Section 7 on how to implement these communications.

The customization service uses its own database to store the data that it cannot save to the standard database. The customization service and the database are deployed in two separate environments, which are in turn hosted by a virtual machine from the same provider as the product environment.

7 Discussions

This section discusses the technical challenges and potential solutions to implement the reference architecture.

7.1 Customization of Database

Customization often needs to extend the standard data type. Two types of extension on the data schema must be supported, i.e., adding a new data entity and adding a field to an existing entity. Removing an entity or a field is not necessary for customization, since the customization service can simply ignore them. Changing the type of a field can be achieved by adding a new field and ignoring the original one. Since the customization service is not allowed to change the data schema of the product database, all data under the extended entity of field has to be stored in the customization database. A new data entity can be implemented as a table in the customization database. A new field can be also implemented as a table in customization database, as a mapping from the primary key of the original table to the extended field.

The customization service registers to the tenant manager how it extends the standard data schema. In this way, the product service knows how each tenant extends its database, so that it can utilize the extended data. For example, *Music.MuTeShop.com* has a page listing all the shopping cart items, originally with price and quantity. When rendering this page, *Music.MuTeShop.com* checks with the tenant manager and gets the information that the customization extends shopping cart items with a new field of donation amount. Therefore, it adds a new column in the shopping cart information table for this field and queries the customization service to fill in this column.

Customization databases usually have simple schema and relatively small amount of data. Therefore, it is reasonable to use light-weight technologies such as PostgreSQL and MySQL. NoSQL database is also a possibility as we have experimented with MongoDB in Section 4.2.

7.2 Triggering of Customization Services

The customization service registers itself to the tenant manager, which allows it to be triggered from one of the predefined extension points in the product service (e.g., as shown in Listing 1). When the control flow reaches this extension point, the product service picks the registered customization service, and *triggers* it. There are two types of triggering, i.e., *synchronous triggering*, when the product service awaits the customization service to finish the triggered logic, and *asynchronous triggering* when it does not.

Synchronous triggering can be implemented as a direct REST invocation from the product service to the customization service. In the product service, the implementation of an extension point can be simplified as an *if-then-else* structure: *if* the product service finds a customization service registered for this point, *then* it invokes this service and continues with the returned value, *else* it executes the standard logic. The more extension points the product service has, the more customization it supports. As an extreme case, the vendor can inject an extension point before each method in the product, using Aspect-Oriented Programming [8]. Synchronous triggering applies to the customization scenarios when the behaviour of the product service has to be influenced by the customization immediately.

Asynchronous triggering can be implemented by the event technology. At an extension point, the product service ejects an event indicating that it has reached this point, together with some context information of the extension point. The event is published to a message queue. If a customization service subscribes this message queue at the right topic, it will be notified by the message queue and triggered to handle this event. The product usually has its internal event mechanism, and therefore, to support asynchronous triggering of customization service, the vendor just needs to publish part of these internal events to the public message queue. Using asynchronous triggering, the customization cannot immediately influence the behaviour of the product service because the control flow of the product service is not blocked by the customization service.

A customization service usually needs both synchronous and asynchronous triggering. Take the visit counting scenario in Section 4.2 as an example, each time an album is visited, the customization service needs to be triggered asynchronously to increase the number of visits in its database. Later on, in the overview page, the product service needs to synchronously trigger the customization service to get the numbers of visits for all the albums. This time it needs to wait for those numbers to be returned from the customization service to show them on the overview page.

7.3 Invocation from Customization Services to the Product Service

A customization service needs to *invoke* the product service, in order to obtain the standard data and to influence the state and behaviour of the product service for the relevant tenant. From a technical point of view, there are two ways of implementing the invocation from customization service to product service, i.e., *intrusive invocation*, when the customization service injects code into the product service, and *non-intrusive invocation*, when the customization service only relies on the APIs opened by the product service.

For intrusive invocation, the customization service send a piece of code (we call it the “callback code“), to the product service. The product service compiles and executes the callback code immediately, and sends the execution results back to the customization service. The callback code is exposed to the same context as the native code of the product service, and therefore, in theory, it can read and write all the standard data and the other states of the product service, just as a piece of native code. To support intrusive invocation, the product should have a built-in interpreter that compiles and executes the callback code. Some modern dynamic programming languages support the execution of source code from plain text, e.g., the `eval` method in Python. Microsoft .Net framework also provides the Dynamic Linq techniques to compile a query into a dynamically executable delegation. If the product is implemented in a platform without such support, the vendors can choose to translate the callback code into a set of invocations to the reflection API. For the sake of security and simplicity, the callback code should be transferred as source code, in terms of plain text, instead of binary code. The product service should provide the specific REST API for injecting callback code and return the execution result.

For non-intrusive invocation, the customization service calls the REST API of product service to obtain the standard data and to manipulate the product states. In this way, the customization capability is defined and limited by the APIs exposed to the customization services. Providing a both powerful and easy-to-use API is a big challenge for the vendor. Automatic generation of such APIs based on the data schema and the product features is a promising way.

One challenge related to the invocation of product instance, regardless of intrusive or non-intrusive way, is how to keep the execution context of the product service. In the product service, every piece of code is running under a runtime context, which is the temporary and static variables accessible by this piece of code. The context contains the important information such as the current user, the recently queried and manipulated data, etc. When the customization service kicks in and replaces an original piece of code, it normally need such context information. For intrusive invocation, a natural solution is to reserve the entire context at the point when the customization service is triggered, and the product service use this context to execute the callback code. For non-intrusive invocation, the vendor should identify the useful context information and provide specific API methods to obtain and exploit such context information.

7.4 Tenant Manager and Tenant Isolation

The tenant manager is a part of the product service, which records the customization for each tenant. When a customization service is activated, it registers to the tenant manager the following information: which tenant it customizes, what extension points it listens to (together with a service endpoint for the product service to call in order to trigger the customization logic), and how it extends the product data schema. The product service queries the tenant manager for such registration every time it reaches an extension point or requires the data extensions. Due to the frequent interaction between the product service and the tenant manager, it is reasonable to implement the tenant manager as a local component within the product service, to avoid the unnecessary overload by remote invocations.

One of the key requirements in multi-tenant SaaS is tenant isolation with security and privacy, especially together with deep customization enabled. We have better addressed this requirement in the non-intrusive approach as discussed in [15]. The non-intrusive approach [15] allows a software vendor to manage all the tenants' customization microservices, in how they are authorized to customize the main product for a specific tenant, in administrating and monitoring the customization microservices at runtime. Deploying customization microservices on separate containers for different tenants and the main product is also very important for tenant isolation as discussed below.

7.5 Customization Environments

A customization environment comprises the infrastructure, technical stack and libraries that a customization service needs at runtime. Considering that each customization service should have a unique isolated environment, and a product service may serve many tenants, a vendor cloud may host a large number of customization environments at the same time. Therefore, it is important to keep a minimal footprint for each customization environment and to simplify the management of these environments.

All the customization environments should use the same type of infrastructure, which is both light-weighted and easy to manage. The container technology, in particular Docker, appears to be very suitable for these purposes. Each customization environment is isolated in a Docker container, so as the environment that hosts a customization database. The consultant provides the vendor a Dockerfile, specifying how to construct a customization environment container, i.e., choosing an operating system, installing the technical stack step by step, downloading the customization App source code, and finally defining how to initialize the technical stack with the customization App. The vendor builds the Docker image according to the Dockerfile, in the vendor cloud, and instantiates a container from the image when customization service needs to be on-board for the tenant.

The vendor should maintain a Docker cluster composing by a flexible number of virtual machines from the same cloud provider. Depending on the number of customization services and the load on them, the vendor cloud should scale in or out of the Docker cluster. The vendor can also apply a standard management tool to monitor the state and resource consumption of each container, and kill the customization services when necessary. Such scaling and management functionality can be implemented using Docker tools.

8 Related Work

Software Product Line (SPL) [18] captures the variety of user requirements in a global variability model, and actual products are generated based on the configuration of the variability model. Traditional SPL approaches target all the potential user requirements by the software vendor, and thus do not apply to our definition of customization. Dynamic SPL [5] is closer to customization, and some approaches such as [10] propose the usage of variability models for customization. However, such model-based configuration is in a much higher abstraction level than programming [20], and focused on how to combine existing features. In contrary, customization is targeting the development of new features specific to the customers.

There are many approaches to SaaS customization in the context of service-oriented computing. However, most of approaches focus on a high-level modification of the service composition. Mietzner and Leymann [13] present a customization approach based on the automatic transformation from a variability model to BPEL process. Here customization is a re-composition of services provided by vendors. Tsai and Sun [24] follow the same assumption, but propose multiple layers of compositions. All the composite services (defined by processes) are customizable until reaching atomic services, which are, again, assumed to be provided by the vendors. Nguyen et al. [16] develop the same idea, and introduce a service container to manage the lifecycle of composite services and reduce the time to switch between tenants at runtime. These service composition approaches all support customization in a coarse-grained way, and rely on the vendors to provide adequate “atomic services” as the building blocks for customized composite services. The microservice architecture discussed in this paper is targeted at how to allow customers to develop the atomic services and integrate them into the product service.

As market leading SaaS for CRM and ERP, the Salesforce platform and the Oracle NetSuite provide built-in scripting languages [21][9][17] for fine-grained, code-level customization. Using these scripting languages, the customization is running within the product, which requires an advanced scripting interpreter to guarantee the isolation between customization and the product. The customization capability is limited by the expression power of the language, and learning these languages is a special burden for the customization developers. Implementing customization as microservices solves these problems: Developers can choose the technical stack that suits them, and still do not need to care about the hosting of these services.

Middleware techniques are also used to support the customization of SaaS. Guo et al. [4] discuss, in a high abstraction level, a middleware-based framework for the development and operation of customization, and highlight the key challenges. Walraven et al. [25] implemented such a customization enabling middleware. In particular, they allow customers to develop customization code using the same language as the main product, and use Dependency Injection to dynamically inject these customization Java classes into the main service, depending on the current tenant. Later work from the same group [26] developed this idea and focused on the challenges of performance isolation and latency of customization code switching. The dependency injection way for customization is close to our work, in terms of the assimilation between custom code and the main service. However, operating the customization code as an external microservice eases performance isolation. A misbehavior of the customization code only fails the underlying container, and the main product only perceives a network error, which will not affect other tenants. Besides, external microservices ease management: scaling independently resource-consuming customization and eventually billing tenants accurately.

This paper is a full extension of the position paper [23]. In this paper, we have summarized our two approaches in [2] and [15] and then presented the full reference architecture for customizing multi-tenant SaaS using microservices. Moreover, we have given discussions on the technical challenges and potential solutions to implement the reference architecture.

9 Conclusion and Future Work

In this paper, we have presented a customization solution for multi-tenant SaaS using microservices. From an intrusive approach, we have evolved our solution to introduce a non-intrusive approach that could be more practical for industry. Based on these two approaches, we have provided a generalized reference architecture for enabling customization of multi-tenant SaaS using microservices. Our discussions on the technical challenges and potential solutions to implement the reference architecture give more insights for readers to adopt our customization solution. Our microservice-based customization solution is promising to meet the general customization requirements, and achieves a balance between isolation, assimilation and economy of scale. Our future research will focus on the quality assurance of the customization services, including automatic testing and online monitoring, to achieve a DevOps way of continuous customization development.

References

1 Cor-Paul Bezemer and Andy Zaidman. Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM, 2010.

2 Franck Chauvel and Arnor Solberg. Using Intrusive Microservices to Enable Deep Customization of Multi-tenant SaaS. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 30–37, September 2018. doi:10.1109/QUATIC.2018.00015.

3 Denise Ganly, Andy Kyte, Nigel Rayner, and Carol Hardcastle. The Rise of the Postmodern ERP and Enterprise Applications World. Gartner Report ID: G00259076, April 2018. URL: https://www.gartner.com/doc/2633315?ref=mrktg-srch.

4 Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. In *e-commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551–558. IEEE, 2007.

5 Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.

6 IDG. 2018 Cloud Computing Survey, April 2018. URL: https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/.

7 Cindy Jutras. Cloud Financials: Having It Your Way, White paper from AICPA. URL: https://online.intacct.com/WebsiteAssets_wp_mintjutras_cloud_financials_your_way.html.

8 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

9 Thomas Kwok and Ajay Mohindra. Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications. In *International Conference on Service-Oriented Computing*, pages 633–648. Springer, 2008.

10 Jaejoon Lee and Gerald Kotonya. Combining service-orientation with product line engineering. *IEEE software*, 27(3):35–41, 2010.

11 Microsoft. eShopOnContainers - Microservices Architecture and Containers based Reference Application. URL: https://github.com/dotnet-architecture/eShopOnContainers.

12 Microsoft. MusicStore test application that uses ASP.NET/EF Core. URL: https://github.com/aspnet/MusicStore.

13 Ralph Mietzner and Frank Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 359–366. IEEE, 2008.

14 Phu H. Nguyen, Hui Song, Franck Chauvel, and Erik Levin. Towards customizing multi-tenant Cloud applications using non-intrusive microservices. *The 2nd International Conference on Microservices, Dortmund*, 2019.

15 Phu H. Nguyen, Hui Song, Franck Chauvel, Roy Muller, Seref Boyar, and Erik Levin. Using Microservices for Non-intrusive Customization of Multi-tenant SaaS. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 905–915, New York, NY, USA, 2019. ACM. doi:10.1145/3338906.3340452.

16 Tuan Nguyen, Alan Colman, and Jun Han. Enabling the delivery of customizable web services. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 138–145. IEEE, 2012.

17 Oracle. Applicaiton Development SuiteScript. URL: http://www.netsuite.com/portal/platform/developer/suitescript.shtml.

18 Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

19 Chris Richardson. Microservices patterns, 2018.

20 Marcus A Rothenberger and Mark Srite. An investigation of customization in ERP system implementations. *IEEE Transactions on Engineering Management*, 56(4):663–676, 2009.

21 Salesforce. Apex Developer Guide. URL: `https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/`.

22 Hui Song, Franck Chauvel, and Arnor Solberg. Deep customization of multi-tenant SaaS using intrusive microservices. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 97–100. ACM, 2018.

23 Hui Song, Phu H. Nguyen, Franck Chauvel, Jens Glattetre, and Thomas Schjerpen. Customizing Multi-Tenant SaaS by Microservices: A Reference Architecture. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 446–448, July 2019. `doi:10.1109/ICWS.2019.00081`.

24 Wei-Tek Tsai and Xin Sun. SaaS multi-tenant application customization. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 1–12, 2013.

25 Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *Proceedings of the 12th International Middleware Conference*, pages 360–379. International Federation for Information Processing, 2011.

26 Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant software-as-a-service applications with service lines. *Journal of Systems and Software*, 91:48–62, 2014.

Experience Report: First Steps towards a Microservice Architecture for Virtual Power Plants in the Energy Sector

Manuel Wickert
Fraunhofer IEE, Kassel, Germany
http://www.iee.fraunhofer.de
manuel.wickert@iee.fraunhofer.de

Sven Liebehentze
Fraunhofer IEE, Kassel, Germany
http://www.iee.fraunhofer.de
sven.liebehentze@iee.fraunhofer.de

Albert Zündorf
University of Kassel, Germany
https://seblog.cs.uni-kassel.de/
zuendorf@uni-kassel.de

Abstract

Virtual Power Plants provide energy sector stakeholders a useful abstraction for distributed energy resources by aggregating them. Software systems enabling this are critical infrastructure and must handle a fast-growing number of distributed energy resources. Modern architecture such as Microservice architecture can therefore be a good choice for dealing with such scalable systems where changing market and regulation requirements are part of every day business. In this report, we outline first experiences gained during the change from the existing Virtual Power Plant software monolith to Microservice architecture.

2012 ACM Subject Classification Software and its engineering → Software design engineering; Hardware → Smart grid

Keywords and phrases Microservices, VPP, Virtual Power Plants, Domain Driven Design

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.2

1 Introduction

In today's transition to green energy, the primary concept of a modern power plant is beginning to become that of a Virtual Power Plant (VPP), consisting of a huge number of small distributed energy resources (DERs) [14]. Usually these DERs are wind farms, photovoltaic parks, biogas plants, energy storages or flexible loads. VPPs have become the modern kind of a large power plant, replacing large conventional power plants over time. The coordination of a VPP is carried out by a software system, often referred to as an energy management system (EMS). Such systems are connected to the distributed energy resources to build the abstraction over a portfolio of DERs and provide monitoring and control capabilities.

During several research projects, Fraunhofer IEE has developed and evaluated such a software system, the VPP software solution IEE.vpp, which is in operation at some utilities and trading companies in the energy sector. The software has with time become a monolith with hundreds of KLOC (kilo lines of code). It consists of a canonical and generic data model [8] which has strengths and weaknesses. In situations where the data model did not fit into the corresponding business logic, the development time for the components increased significantly. In the energy domain, where the regulatory framework changes very often and new business models have to be implemented quickly, this will become a bigger issue over

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 2; pp. 2:1–2:10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

time. Furthermore, it is hard for domain experts with algorithmic skills to contribute to the VPP solution because their main programming languages are Python and Matlab, but the VPP solution is written in Java. Therefore, we decided to migrate our macro architecture to a Microservice approach [5, 12], based on a Domain-Driven Design [4].

With the migration of our software systems we evaluate two different aspects. On the one hand, we outline how the modern architecture style of Microservices can be applied to the domain of VPPs. While there is currently some work on evaluating Microservices architectures in other areas of the smart grid such as metering [11] or IoT (Internet of Things) for smart buildings [2], most publications [16, 9, 10] for VPPs do not consider Microservices. Thus we present a first step for the application of Microservices as an architectural style for VPPs that should be transferable to other VPP software systems. On the other hand, using the example of a VPP, we show a novel migration approach to a different technology (e.g. programming language). For the reasons stated above, we planned to implement some Microservices in Python instead of using only Java.

This report presents the first steps of our migration to a Microservice architecture. The migration was done step-by-step during an agile development process to enable us to provide our customers with regular updates. The first step was a rough analysis of our domain to identify the main business functions to be supported by our VPP software system. Section 2 gives an overview of our Analysis. With this information, we are able to identify different bounded contexts for our software system in Section 3. For one of these bounded contexts, the migration to our first Microservice is described step by step in Section 4. In Section 5 we present the results of our first migration steps and discuss the advantages and disadvantages of our approach. Section 6 contains our conclusions, points out next steps as well as our most remarkable achievements.

2 Analyzing the Domain

From the perspective of Domain-Driven Design, VPPs typically use the two different domains of energy trading and grid operation in the energy sector. In [3, 14], for these domains, two different types of virtual power plants have been introduced - the technical virtual power plant (TVPP) and the commercial virtual power plant (CVPP). Although some software systems try to support use cases in both domains, most VPPs focus on just one. For the European market this will be due to the fact that the liberalization of the energy markets in Europe stipulates that TVPPs and CVPPs have to be operated by different companies and therefore have to support different business models. The VPP IEE.vpp software solution is a CVPP energy management system. Thus, the next sections focus on domain energy trading.

To understand the energy trading domain, we start with an example. The trading company “Green Trader” is highly skilled in renewable energy trading, for DERs dealing in windfarms, photovoltaics, biogas plants, batteries etc. For this reason it buys energy from different power plant owners, paying a price fixed by contract for the power fed into the grid. The fixed price is typically negotiated for one year. The “Green Trader” company bundles all bought energy in one portfolio and trades it on the EPEX SPOT energy market. Because the energy has to be traded before delivery, wind and photovoltaic production is forecasted and the current generation is monitored. Differences between traded and delivered energy is subject to a penalty payment for the trader. For optimizing the DER behaviour, according to market prices and to avoid penalty payments due to forecast errors, the “Green Trader” company is able to control the production and consumption (e.g. for batteries) of the DERs. To summarize, the “Green Trader” company needs to be able to monitor and control all DERs and optimize their schedules according to market prices. Therefore the company uses an energy management system for VPPs, such as the IEE.vpp system.

This example highlights two important requirements, “Monitoring and Control” and “Optimization” for an energy management system of a VPP from a trading perspective. However energy trading is much more complex and requires a number of different software systems. Therefore the energy management system should also support the provision of the collected information concerning the DERs, which is used by third party systems such as market clients, process and data management tools, etc. This information provision may be also needed for communication with other companies such as forecast providers. The provision of information may be a use case for many other software systems as well. However, since EMS hold critical business information concerning the DERs, information provision to another system is a crucial functionality. Therefore, we highlight this requirement as well.

The portfolio of energy traders changes over time, in Germany every year. Thus the integration of new energy units in a VPP and the termination of existing connections is a permanent topic. In addition to base data management, the integration of a DER consists primarily of the establishment of a communication link. The configuration of different communication protocols and manufacturer‘ specific data models is also one of the primary features of an EMS. Over the previous few years in many meetings with about 30 percent of all German energy traders, we became aware that the easy integration of power plants is one of the most desired requirements when building a VPP.

The analysis of the typical use cases for energy trading led to the following four main business functions that need to be supported by our EMS:

- Monitoring and Control
- Flexibility Optimization
- Information Provision
- Administration of DER

3 Identifying Bounded Contexts

Based on the described use case analysis and the subsequent designated business functions, we were able to define the domain precisely and derive the bounded contexts [12, 5, 15]. To outline the dependencies between these, we then built a context map [4]. Please note, that the business function “Information Provision” was skipped during this analysis because it can be regarded as a more abstract business function. The main task of this business function is the provision of information to third party systems. In order to define the associated domain model, the external interface specification must be known. These reasons led to three bounded contexts and a context map. In the context map we outline how we can integrate concrete bounded contexts for information provision.

The first bounded context “SCADA” (Supervisory Control And Data Acquisition) includes the supervision of control and data aquisition in the VPP software system, see Figure 1. A DER is able to read different sensor values. These might be measurements such as the current active power production, different temperatures or wind velocities. On the other hand a DER might be able to process control commands, for example to reduce the current power production. In order to exchange information and control commands between the VPP and the control system of a DER, a certain protocol is used as a rule, for example OPC-UA or IEC 60870-5-104. A so-called mapping scheme defines the translation between specialized protocol items or addresses and the VPP internal representation.

The next identified bounded context is the so-called “Aggregation”, see Figure 2. It includes aggregation and disaggregation data management strategies regarding the respective energy market commitment of the DERs. The Aggregation handles the management of portfolios, which are part of the VPP. Each portfolio is split according to various criteria, such as plant type or marketing strategies.

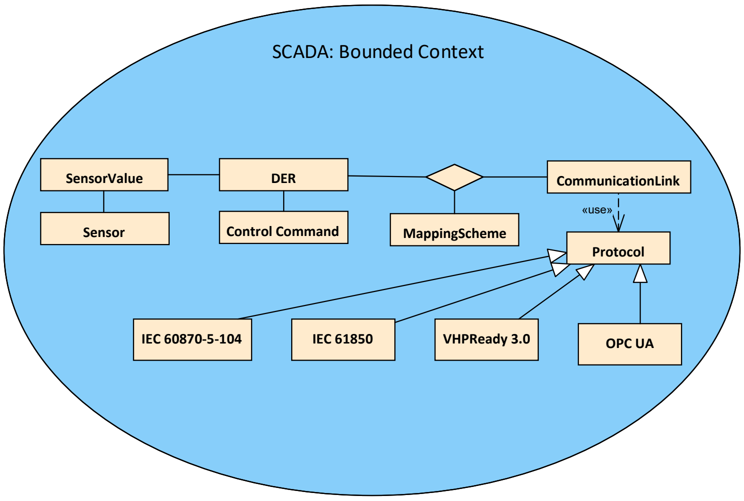


Figure 1 Bounded Context SCADA.

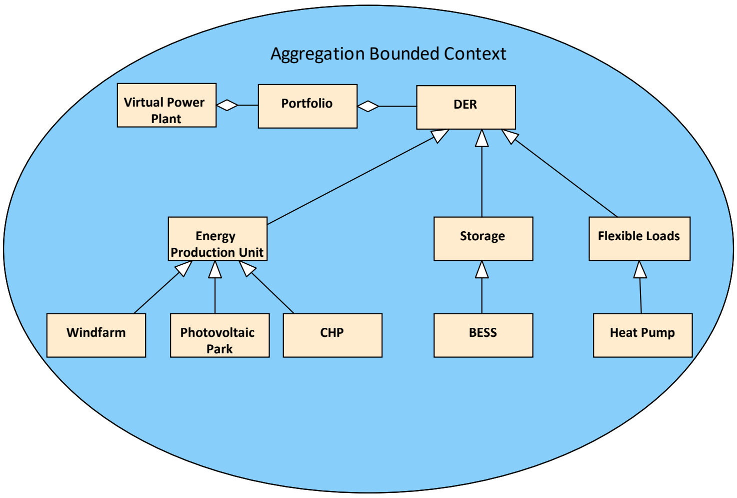


Figure 2 Bounded Context Aggregation.

The last identified bounded context is needed to model the unit commitment of the DERs. It is called “Optimization”, see Figure 3. A portfolio may be placed on one or more markets. Corresponding electricity price forecasts support the optimal electrical schedule calculation for the related plants. Figure 3 shows part of the domain model for the combined heat and power (CHP) optimization. For the best operation of a CHP, the energy production costs are mainly determined by the fuel costs, i.e. gas prices. Furthermore, the efficiency curve is taken into account in calculating how much electrical and thermal energy is produced depending on the amount of gas. Heat storage is another important influencing factor because it cannot be overfilled. The heat sink usually models the tight restrictions for the thermal energy consumption.

With the three defined bounded contexts we are able to realize the business functions from Section 2, except for the “Information Provision” as already noted. For the “Monitoring and Control” business function, the “SCADA” and “Aggregation” contexts are necessary. This is because the “SCADA” context contains the base model for control and monitoring. The

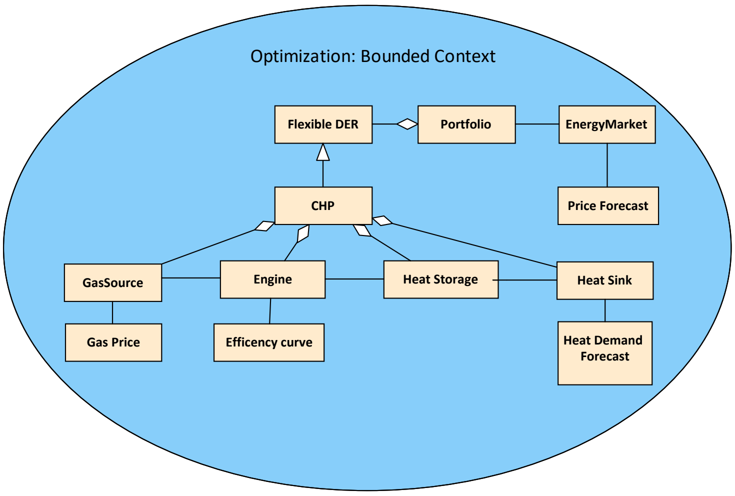


Figure 3 Bounded Context Optimization.

"Aggregation" context helps us to monitor different levels of aggregation and to disaggregate control signals. The "SCADA" context supports the "Administration of DER" business function as well. The domain model for the "Flexibility Optimization" is represented by the bounded context "Optimization" .

The three described bounded contexts and an abstract bounded context "Information Provision" are shown together in the context map in Figure 4. The Mappings between the bounded contexts are shown by the use of the patterns for strategic design, cf. [4]. We use the patterns Open Host Service (OHS) and Anticorruption Layer (ACL) for our purpose.

The "SCADA" context shares part of its model for providing sensor values and receiving control commands, the "Aggregation" context uses this OHS but translates it with the use of an ACL. The "Optimization" context uses processed information in a certain time resolution e.g. 15 minutes mean values. This information is provided by the bounded context "Aggregation" . The optimized schedules for the power plants may send directly to the "SCADA" context. However also schedules for the optimization have to be aggregated for the portfolios. In the current design, we plan to only have a link between "Aggregation" and "Optimization" . There is still the option to also establish a link between "Optimization" and "SCADA" . The context map also shows an abstract bounded context "Information Provision" . Typically this bounded context will receive information through a shared model from the "Aggregation" context and will use an ACL for the translation to the specific model.

4 Architectural Migration

We approached the migration case by case, starting with the bounded context Optimization. As already noted, in order to include the domain experts in the respective area of optimization, i.e. Mixed Integer Linear Programming (MILP) [13], we decided to implement the new Microservice in Python. Additionally, this enabled the use of popular MILP frameworks such as Pyomo [7, 6].

Before the migration, we identified parts of the old monolithic architecture, which are responsible for the Optimization (marked in yellow in Figure 5). Therefore a closer look at the monolithic architecture was needed. The software is implemented as a classic client server architecture, where the backend is built as a layered architecture (see 1. in Figure 5).

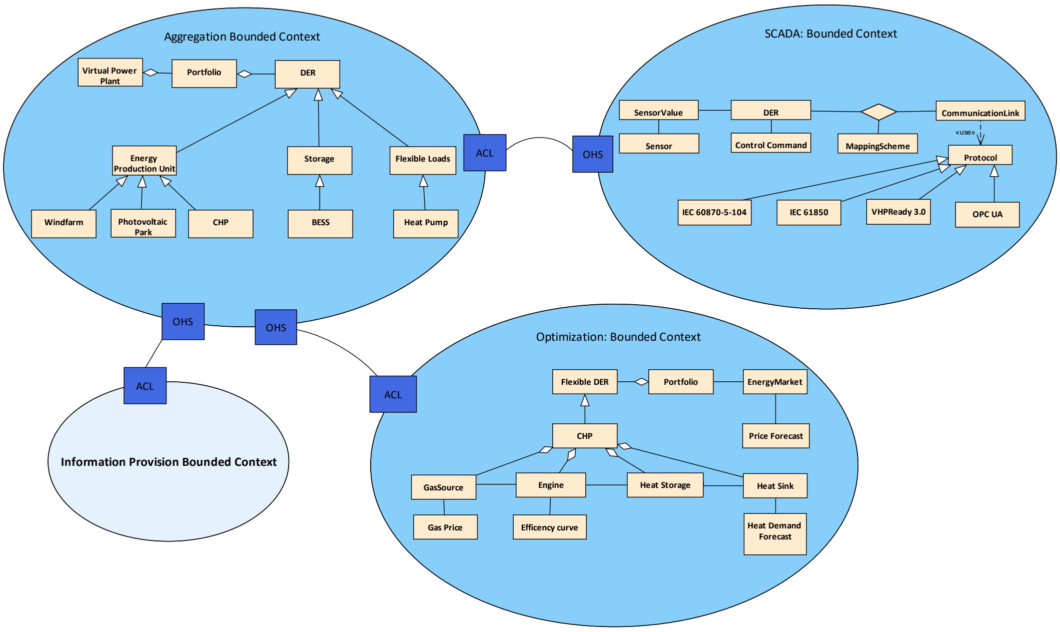


Figure 4 Context Map.

The user interface is written in JavaFX and implemented as a rich client. It connects via RESTful HTTP endpoints with the service layer of the backend. The service layer routes requests to the domain logic, e.g. the optimization kernel. The business logic uses raw data from DERs as well as aggregated and interpolated information. For this reason the business layer above an aggregation and disaggregation layer. The bottom layer consists of online data information retrieved directly from DERs as well as historical data retrieved from the database. In 1. of Figure 5 the parts that are responsible for the optimization are highlighted (marked in yellow). They are distributed over the different layers as well as in the UI and database.

In the next step (2. in Figure 5) the optimization kernel was implemented in Python. In contrast to a classical migration within the same programming language, a refactoring based migration, cf. [12], was not possible. To continue working within our agile development process and to keep the customer up to date, we decided not to extract all optimization related functionalities at this stage. Therefore we only extracted the core business logic into a SOA-like service and called these services directly from the layer where they were extracted from. In this step, we used the data model from the above defined bounded context and we used the OHS and ACL patterns, cf. [4], to transform the information between the monolith and the new service. The other layers, e.g. persistence and aggregation, as well as the UI, remained in the monolith. After this step, all existing integration and contract tests from the monolith could be used to verify that the VPP software worked as before.

In step three, we enhanced the SOA-Service to a Microservice and shifted the relevant parts of the aggregation and persistence layer from the monolith while adding a new database to the service as well (see 3. of Figure 5). Introducing a new user interface or a micro-frontend was skipped during this step. The main reason for this was the existing UI Technology. For JavaFX clients, there are no appropriate micro-frontend approaches which are independently deployable. Besides our Microservice migration we worked on the development of a web UI, therefore we shifted the UI separation to another step. To keep providing a common

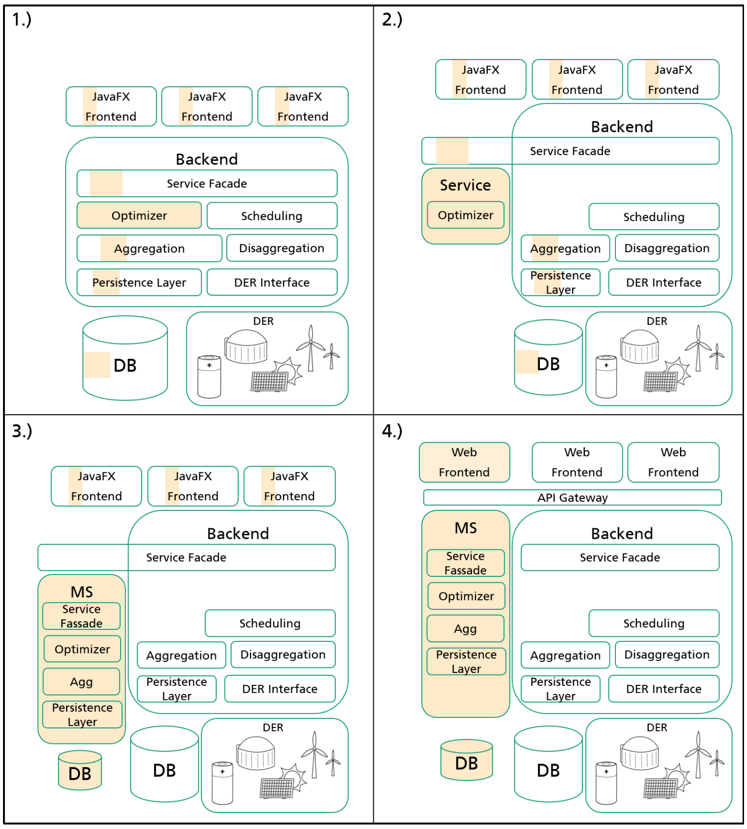


Figure 5 Four migration steps including the highlighted use of the bounded context Optimization.

interface to the UI and other applications that directly communicate with the REST service, we needed some kind of API Gateway. To minimise infrastructure changes for the first Microservice, we used the existing service facade as the API Gateway for the monolith and the Microservice as well.

The last step is to introduce a web UI and a separate API Gateway. However this step is only useful if the UI is completely based on a web UI. Currently the migration of the UI to Angular [1] is ongoing. After finishing this we will decide which existing API Gateway solution we are going to use. With the completion of this step, the full migration is finished and independent deployment will be possible.

The Architectural Migration was performed in four separate steps during our SCRUM-based development process with 2 teams (see Section 5). We had three-week iterations and each step took several iterations because the migration was done in parallel to the feature development. Overall it took about six months to deliver the first deployment to our customer. The reason for this being that we still needed some iterations in order to implement the existing functionality in Python. Also the integration of the domain experts into the new development Team took some time. For each of the migration steps we could perform existing contract tests, partial integration tests and deploy our software in production on schedule.

During our migration we identified two main challenges. The first was to bring the python modules into operation including logging and metrics, providing health status, evaluating web service frameworks and safety and security considerations, etc. The team's goal was to provide at least the same stable operation behavior as our Java Software already had. The second challenge was to implement component tests. Now, the mocking of the optimization changed from mocking Java Components to mocking a Microservice or an external Interface. Thus the tests scenarios became much more complex.

5 Results

As a result of the first migration we received a smaller monolith (about 270 KLOC) and a new Optimization Microservice (about 10 KLOC). Consequently we could significantly increase the number of developers for the product by the inclusion of our domain experts. We started with one development team for the monolith with 6 developers and have now two development teams with 5 (allmost all well experienced in software engineering) and 4 developers (1 software engineer, 3 economists/industrial engineers). However, the new development team has more skills in linear programming than in software engineering. Thus the implementation of the agile development process is not yet complete.

The deployment of the software has also changed. To frequently deliver updates we changed our production infrastructure to Docker together with the infrastructure team of our customers. This helped us to deploy Python software including all required modules in a stable environment. It also separated the infrastructure dependencies of the monolith and the new Microservices. A consequence of this is that the infrastructure is now more complex than before. To cope this, we are now working on the implementation of a distributed logging environment.

The migration itself gained from the fact that we implemented a Service oriented architecture (SOA) in the second step. This focused our implementation on the domain logic parts of the Microservice and without a database the infrastructure was less complex in the second step. In comparison with a direct Microservice implementation, our approach needs more development time because the integration consists of more steps. For working in agile development teams we still recommend our new approach for migrating to a different programming language, also for completely different domains.

6 Conclusion

In this experience report, we analyzed the domain of VPPs and pointed out relevant business functions for a CVPP. For each business function, we identified supporting models and built bounded contexts with a context map. With this domain-driven approach, we started the migration with the detachment of the bounded context Optimization. With this step a new Microservice evolved.

For the architectural migration, we examined a novel approach which separates the process into four steps. This approach is also transferable to other software systems that intend to use a new programming language within a new Microservice. Our most important achievement was the realization of the respective upcoming migration step while delivering new software features at regular intervals. The separation of the last step was important due to the existing UI technology. This may also be an issue for other existing systems.

Our greatest benefit was the first migration step, namely the inclusion of further expert knowledge in the development team by using Python as the programming language of the new Microservice. This also opened up the set of actively supporting development team members within our organization. Another benefit is the sharper domain within the new Microservice supporting a more rapid development cycle for optimization features.

The next step will be to also perform the migration of the user interface to a pure Web UI. With this technology it is possible to build so called Microfrontends and deploy them together with the Microservice. This also enables us to postpone UI framework or library decisions to development teams. Afterwards we will be able to start the decomposition of the remaining monolith to a SCADA and an administration service. A further decomposition may follow afterwards. One of the future challenges will be the measurement of the effect of the Microservice migration in terms of scaling behavior in operation, shorter development times, etc. This will give us indicators for further architectural decisions.

References

1 Angular Framework. URL: `https://angular.io/`.

2 Kaibin Bao, Ingo Mauser, Sebastian Kochanneck, Huiwen Xu, and Hartmut Schmeck. A Microservice Architecture for the Intranet of Things and Energy in Smart Buildings: Research Paper. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA@Middleware 2016, Trento, Italy, December 12-13, 2016*, pages 1–6, December 2016. `doi:10.1145/3007203.3007215`.

3 Martin Braun. Virtual power plants in real applications: Pilot demonstrations in Spain and England as part of the european project FENIX. *Fraunhofer IWES*, January 2009.

4 Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

5 Martin Fowler and James Lewis. Microservices, 2014. URL: `http://martinfowler.com/articles/microservices.html`.

6 William E. Hart, Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, second edition, 2017.

7 William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

8 G. Hohpe and B.A. WOOLF. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Prentice Hall, 2004. URL: `http://books.google.com.au/books?id=dH9zp14-1KYC`.

9 B. Jansen, C. Binding, O. Sundstrom, and D. Gantenbein. Architecture and Communication of an Electric Vehicle Virtual Power Plant. In *2010 First IEEE International Conference on Smart Grid Communications*, pages 149–154, October 2010. `doi:10.1109/SMARTGRID.2010.5622033`.

10 Ingo Mauser, Christian Hirsch, Sebastian Kochanneck, and Hartmut Schmeck. Organic Architecture for Energy Management and Smart Grids. In *2015 IEEE International Conference on Autonomic Computing, Grenoble, France, July 7-10, 2015*, pages 101–108, 2015. `doi:10.1109/ICAC.2015.10`.

11 Antonello Monti, editor. *Technologies and Methodologies in Modern Distribution Grid Automation*, volume 15. Elsevier, Amsterdam [u.a.], 2018. URL: `http://publications.rwth-aachen.de/record/745831`.

12 Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

13 Seyyed Mostafa Nosratabadi, Rahmat-Allah Hooshmand, and Eskandar Gholipour. A comprehensive review on microgrid and virtual power plant concepts employed for distributed energy resources scheduling in power systems. *Renewable and Sustainable Energy Reviews*, 67(C):341–363, 2017. `doi:10.1016/j.rser.2016.09.02`.

14 H. Saboori, M. Mohammadi, and R. Taghe. Virtual Power Plant (VPP), Definition, Concept, Components and Types. In *Proceedings of the 2011 Asia-Pacific Power and Energy Engineering Conference*, APPEEC '11, pages 1–4, Washington, DC, USA, 2011. IEEE Computer Society. `doi:10.1109/APPEEC.2011.5749026`.

15 Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st edition, 2013.

16 Matej Zajc, Mitja Kolenc, and Nermin Suljanovic. *Virtual Power Plant Communication System Architecture*, chapter 11, pages 231–250. Elsevier, 2018. doi:10.1016/B978-0-12-812154-2.00011-0.

Exploring Maintainability Assurance Research for Service- and Microservice-Based Systems: Directions and Differences

Justus Bogner
University of Applied Sciences Reutlingen, Herman Hollerith Center, Germany
University of Stuttgart, Institute of Software Technology, Software Engineering Group, Germany
https://www.hhz.de/en/research/research-groups/digital-enterprise-architecture
justus.bogner@iste.uni-stuttgart.de

Adrian Weller
University of Stuttgart, Institute of Software Technology, Software Engineering Group, Germany
https://www.iste.uni-stuttgart.de/se
adrian.weller94@gmail.com

Stefan Wagner
University of Stuttgart, Institute of Software Technology, Software Engineering Group, Germany
https://www.iste.uni-stuttgart.de/se
stefan.wagner@iste.uni-stuttgart.de

Alfred Zimmermann
University of Applied Sciences Reutlingen, Herman Hollerith Center, Germany
https://www.hhz.de/en/research/research-groups/digital-enterprise-architecture
alfred.zimmermann@reutlingen-university.de

Abstract

To ensure sustainable software maintenance and evolution, a diverse set of activities and concepts like metrics, change impact analysis, or antipattern detection can be used. Special maintainability assurance techniques have been proposed for service- and microservice-based systems, but it is difficult to get a comprehensive overview of this publication landscape. We therefore conducted a systematic literature review (SLR) to collect and categorize maintainability assurance approaches for service-oriented architecture (SOA) and microservices. Our search strategy led to the selection of 223 primary studies from 2007 to 2018 which we categorized with a threefold taxonomy: a) architectural (SOA, microservices, both), b) methodical (method or contribution of the study), and c) thematic (maintainability assurance subfield). We discuss the distribution among these categories and present different research directions as well as exemplary studies per thematic category. The primary finding of our SLR is that, while very few approaches have been suggested for microservices so far (24 of 223, ∼11%), we identified several thematic categories where existing SOA techniques could be adapted for the maintainability assurance of microservices.

2012 ACM Subject Classification Software and its engineering → Software evolution; Software and its engineering → Maintaining software; Social and professional topics → Quality assurance; Information systems → Web services

Keywords and phrases Maintainability, Software Evolution, Quality Assurance, Service-Based Systems, SOA, Microservices, Systematic Literature Review

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.3

Supplement Material https://github.com/xJREB/slr-maintainability-assurance

Funding *Justus Bogner*: Funded by the Ministry of Science of Baden-Württemberg, Germany, for the doctoral program *Services Computing* (https://www.services-computing.de/?lang=en).

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 3; pp. 3:1–3:22
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

While software continues “to eat the world” [3], it becomes all the more important that systems can be quickly adapted to new or changed requirements. As more and more business processes are not only supported by software, but digital goods and services form the essence of entire businesses, the sustainable evolution of the underlying software is a vital concern for many enterprises. The associated quality attribute is referred to as *maintainability* [35], i.e. the degree of effectiveness and efficiency with which a system can be modified. In fast-moving markets, the adaptive and extending notion of this quality attribute – also referred to as *evolvability* [68] – is especially crucial. To address this, software professionals rely on a set of diverse activities aiming to ensure a sufficient degree of maintainability. We refer to these approaches and techniques as *maintainability assurance*. In general, such activities are either of an *analytical* nature, i.e. to identify existing issues, or of a *constructive* nature, i.e. to remediate or prevent issues [79]. Examples for analytical techniques include metric-based evaluation (both static and dynamic analysis), scenario-based evaluation, or code review. Examples for constructive techniques are refactoring, standardization, or systematic maintainability construction with design patterns. Furthermore, maintainability assurance for larger systems is often structured into a communicated assurance process and is an integral part of the development flow.

The introduction of service-oriented computing [59] arguably led to several maintainability-related benefits such as encapsulation, strict separation between interface and implementation, loose coupling, composition, and reuse. The two service-based architectural styles – namely *service-oriented architecture* (SOA) [23] and *microservices* [53] – are very popular for implementing enterprise applications or web-based systems with strong requirements for scalability and maintainability. The younger microservices paradigm also places special emphasis on evolutionary design [29]. Several publications have tried to summarize the differences and commonalities between SOA and microservices [66, 89, 21, 12]. While no holistic consensus has been reached so far (and probably never will be), many authors focus on the broad set of architectural commonalities. Highlighted differences of microservices are e.g. their decentralized governance and organization (as opposed to centralization and standardization in SOA), their focus on very few lightweight communication protocols like RESTful HTTP (as opposed to protocol-agnostic interoperability via an enterprise service bus in SOA), or their “share nothing” principle (as opposed to SOA’s focus on abstraction and reuse). Nonetheless, the majority of publications acknowledges the shared service-oriented principles like loose coupling, location transparency, or statelessness. Early adopters from industry like Netflix also referred to their system as “fine-grained Service Oriented Architecture” [80].

In principal, both SOA and microservices are based on beneficial properties for maintainable and evolvable systems. However, concrete maintainability assurance processes for such systems are still not trivial to establish. Empirical studies about industry practices in this regard also highlighted that there is a high trust in the base maintainability of service orientation, which may even lead developers to actively reduce assurance efforts [78, 9]. Simultaneously, practitioners are uncertain how to handle service-oriented particularities in this regard and especially report challenges for architectural evolvability [9, 10]. When trying to get an overview of assurance approaches for service orientation proposed by academia, it is not easy to quickly scan the scattered variety of existing publications. Researchers have suggested a plethora of assurance techniques specifically designed for SOA, microservices, or both that try to approach maintainability from different directions. To enable such an overview, we therefore conducted a systematic literature review (SLR) to collect and categorize existing maintainability assurance research for service orientation. Our review explicitly included both SOA and microservices, because the number of relevant publications for the younger architectural style would have been too small on its own. Moreover, we believed that a lot of the approaches designed for SOA will have relevance for microservices as well. Lastly, the inclusion of both service-based styles will enable an additional comparison on this level. Our SLR was guided by the following four research questions:

- **RQ1:** How can maintainability assurance research proposed for service- and microservice-based systems be categorized?
- **RQ2:** How are the identified publications distributed among the formed categories?
- **RQ3:** What are the most relevant research directions per identified category?
- **RQ4:** What are notable differences between the approaches proposed for service-based systems and those for microservices?

In the remainder of this paper, we first present related literature studies (Section 2) and describe the details of our own study design (Section 3). After that, we present the SLR results, from which we synthesized the answers to our research questions (Section 4). Lastly, we mention possible threats to validity (Section 5) and conclude with a summary as well as an outlook on potential follow-up research (Section 6).

2 Related Work

Several existing literature studies cover maintainability-related aspects without focusing on a specific system type or architectural style. Breivold et al. [14] conducted an SLR to collect studies on the architectural analysis and improvement of software evolvability. They identified 82 primary studies that they structured into five categories, such as quality considerations during design, architectural quality evaluation, architectural knowledge management, or modeling techniques. Service-oriented approaches are not included.

Similarly, Venters et al. [77] synthesized existing research approaches for general architecture sustainability in a non-systematic review. They define sustainability as a system's capacity to endure. Service-oriented approaches are only briefly mentioned in the area of reference architectures, where some proposals for SOA are listed. Other described topics are the importance of architectural decisions or metrics for the quantification of sustainability.

There are also several service-based literature studies focusing on SOA. Back in 2009, Gu and Lago [33] conducted an SLR to uncover general service-oriented system challenges. Using 51 primary studies, they identified more than 400 challenges, most of them related to quality attributes, service design, and data management. Only three reported challenges were associated with maintenance, i.e. their review is broader than our intended scope.

A more fine-grained scope than the one intended by us was applied in the literature review of Bani-Ismail and Baghdadi [6]: they solely focused on service identification (SI) in SOA and derived eight different challenges for this activity. The maintainability-related aspect of service granularity is presented as one of the most important challenges.

In another service-oriented SLR, Sabir et al. [69] analyzed the evolution of object-oriented and service-oriented bad smells as well as differences with their detection mechanisms. From 78 publications, they identified 56 object-oriented and 19 service-based smells and presented details about their detection approaches. Smells related to microservices are not covered.

In similar fashion, Bogner et al. [8] conducted an SLR to collect existing antipatterns for both SOA and microservices. While they did not include many details on detection approaches, they synthesized a holistic data model for all antipatterns and also created a taxonomy for their categorization. 14 of the 36 antipatterns were categorized as applicable to SOA, three to microservices, and 19 to both styles. Like the review of Sabir et al., this is a subset of our intended scope, but nonetheless targets both SOA and microservices.

Several more recent literature studies also focus exclusively on microservices. Di Francesco et al. [20] used a systematic mapping study to create a research overview on architecting with microservices. They derived a classification framework and used it to produce a systematic map of the topics of 103 selected primary studies. While maintainability is mentioned in 43 studies as an important design goal or investigated quality attribute, the broad scope of the review prevents a more detailed discussion of how maintainability is actually ensured.

Lastly, Soldani et al. [73] systematically surveyed the existing grey literature on microservices to distill their technical and operational "pains and gains". Afterwards, identified concerns were assigned to common stages of the software life cycle such as design or operation. Maintainability is briefly discussed as an advantageous "gain" based on small service size and self-containment, but concrete techniques for its assurance are not mentioned.

In summary, none of the presented related studies focus exclusively on maintainability and its assurance while simultaneously targeting service- and microservice-based systems. The studies that have a similar scope do not limit their review to service orientation and the ones that do are either too general or too specific in their discussed aspects. We intend to close this gap by presenting an SLR that specifically analyzes the state of the art of maintainability assurance for service- and microservice-based systems.

3 Research Methodology

In general, an SLR is a secondary study that is used to identify, analyze, and summarize (scientific) publications within a certain research area of interest. As such, it presents an overview of the state of the art in a certain (sub)field and may point out research gaps or even a research agenda to close them. Since scientific rigor and replicability are very important in such studies, we relied on the process and guidelines described by Kitchenham and Charters [39]. Moreover, we published all research artifacts in an online repository[1].

Our general research process for this study was as follows (see also Fig. 1). First, we brainstormed about research questions we intended to answer and defined four questions that built upon each other (see Section 1). As a second step, we designed a detailed protocol to guide us through the review. This protocol contained the used data sources (literature databases and search engines), a search term with keywords, filter criteria for inclusion and exclusion of studies, as well as a description of the process. We then used this protocol to retrieve the initial result set from all data sources and subsequently applied our filter criteria. The first two authors individually analyzed and filtered all identified publications and afterwards compared the results. Any differences were discussed until a consensus was reached. For the remaining studies, we performed one round of forward citation search ("snowballing") and applied the same filter criteria to the newly identified publications (again with two researchers). Included studies were merged into the existing set and duplicates were removed. This final set of primary studies was now analyzed in an iterative process. A categorization scheme was derived and subsequently applied to the publications. The result was then analyzed again and possible improvements for the categorization scheme were implemented, which led to the next round of categorization. As with the inclusion and exclusion criteria, categorization of the whole set was performed by two researchers, who discussed any difference of opinion. Once the categories were stable, we started the detailed evaluation to synthesize the answers to our research questions.

[1] https://github.com/xJREB/slr-maintainability-assurance

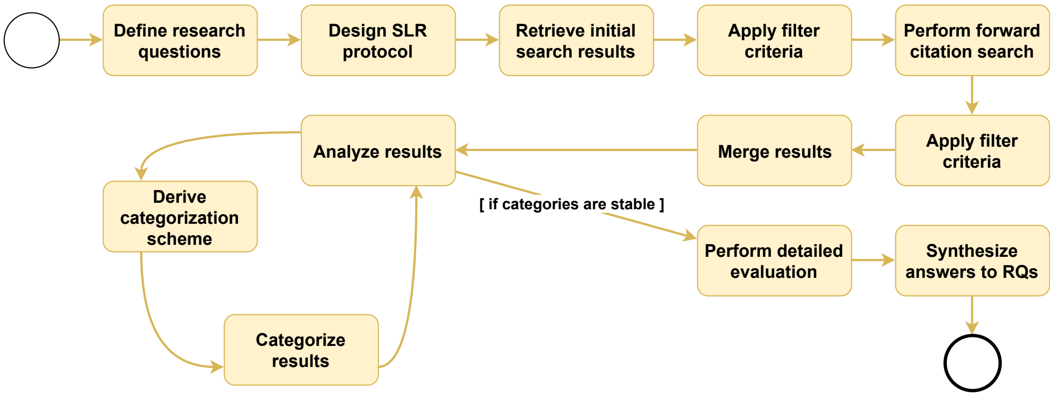

Define research questions
Design SLR protocol
Retrieve initial search results
Apply filter criteria
Perform forward citation search
Apply filter criteria
Merge results
Analyze results
Derive categorization scheme
Categorize results
[if categories are stable]
Perform detailed evaluation
Synthesize answers to RQs

Figure 1 General Research Process.

The four used data sources for our initial search (see also Fig. 2) were IEEE Xplore, ACM Digital Library, Springer Link, and ScienceDirect, as they are very common for software engineering and service-oriented topics. For the snowballing phase, we relied on the publisher-agnostic search engine Google Scholar.

- IEEE Xplore: `https://ieeexplore.ieee.org`
- ACM Digital Library: `https://dl.acm.org`
- Springer Link: `https://link.springer.com`
- ScienceDirect (Elsevier): `https://www.sciencedirect.com`
- Google Scholar (only for snowballing): `https://scholar.google.com`

Figure 2 Used Digital Libraries and Search Engines for the SLR.

As our search string (see also Fig. 3), we formed two buckets with keywords. The two buckets were combined with an `AND` relation while the keywords within each bucket were combined with an `OR` relation, i.e. from each bucket, at least one term needed to match. The first bucket contained our central quality attribute `maintainability` as well as the closely related terms `modifiability`, `evolvability`, and `evolution`. The second bucket was responsible for our targeted system types and therefore consisted of the terms `soa`, `microservice`, `service-oriented`, and `service-based`. The search string was not confined to any particular field.

```
(maintainability ∨ modifiability ∨ evolvability ∨ evolution) ∧ (soa
∨ microservice ∨ service-oriented ∨ service-based)
```

Figure 3 Used Search String for the SLR.

Since we only relied on manual filtering to avoid the accidental exclusion of fitting studies that just use different keywords, we also had to limit the result set to a manageable amount. We therefore only considered the first 250 results per data source, i.e. we had a total of 1000 publications for manual analysis. As our most basic inclusion criteria, we only considered publications written in English and published in the years 2007 up until 2018.

The title and abstract of studies passing this test were then assessed for their relevance to our research questions. The main focus of the paper needed to be on analyzing or improving maintainability (or a related quality attribute or design property) in the context of service-oriented computing (e.g. SOA or microservices). For example, the architecture sustainability review of Venters et al. [77] fulfills the first property, but is not primarily about service orientation. Conversely, the SOA policy optimization approach from Inzinger et al. [36] is clearly about service orientation, but does not solely target maintainability. If the topic relevance could not be determined from title and abstract alone, other parts of the paper like the introduction or conclusion had to be read. Finally, we excluded the fields of runtime adaptation as in [26], software testing as in [37], and legacy to SOA or microservices migration as in [47]. While these topics are related to maintenance and evolution, they are very specialized and each one could probably provide enough material for a separate SLR.

4 Results

Using the process described above, we obtained an initial set of 1000 papers, i.e. 250 per selected publisher. We then applied our inclusion criteria, which resulted in a filtered set of 122 papers. In the snowballing phase, we identified a total of 806 publications that cited a paper from our filtered set. Lastly, we applied the same filter criteria to these new publications and merged the remaining ones back into the filtered set while removing duplicates. This resulted in a final set of 223 primary studies (see also Fig. 4). Nearly half of these papers (105) were published by IEEE, followed by 40 Springer publications (18%), 31 papers from ACM (14%), and 10 from ScienceDirect (4%). The remaining 37 publications (17%) were from 31 different publishers (see also Fig. 5). When looking at the number of publications per year (Fig. 6), we see a slow beginning in 2007 (six publications), a peak in 2011 (35 publications), smaller yet fairly similar numbers for 2012 to 2015 (22-24 publications), and finally another decline for 2016 to 2018 (15-18 publications).

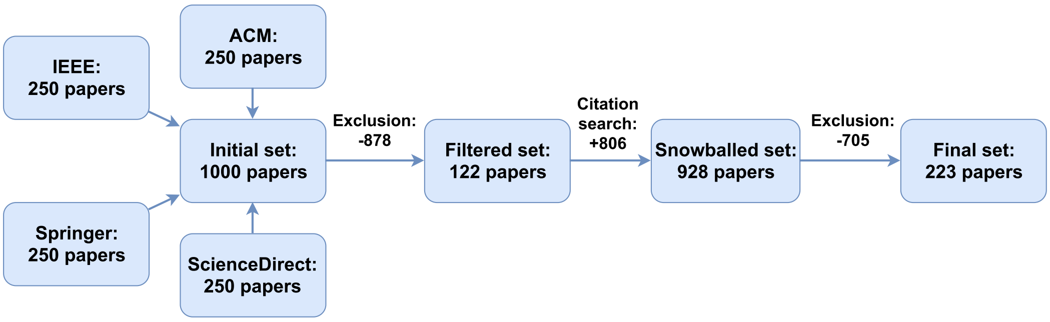


Figure 4 SLR Results: Number of Publications per Stage.

4.1 Research Categories (RQ1)

To answer the first research question, we derived a three-dimensional taxonomy to categorize the identified primary studies (see Table 1). The first and most obvious category type called *architectural* consisted of three different categories that determined if a study targeted *SOA*, *microservices*, or *both* architectural styles. *Both* was selected if the study either explicitly stated the inclusion of both SOA and microservices or if it was about concepts like RESTful services that are prevalent in both styles. This category type was mandatory and exactly one

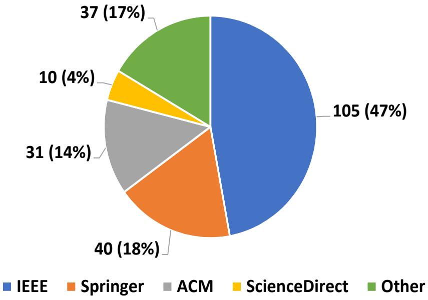


Figure 5 Publisher Distribution.

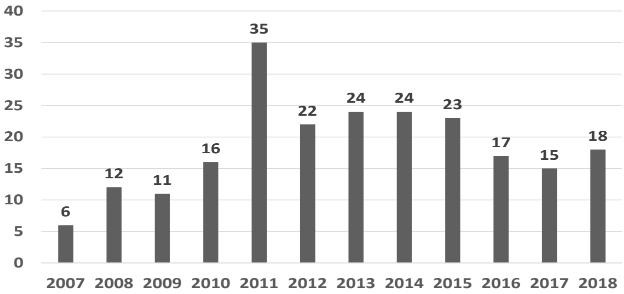


Figure 6 Number of Publications per Year.

Table 1 Three-Dimensional Categorization Scheme.

Type	Description	Mandatory	Multiple
Architectural	Contains three categories that determine if a publication focuses on *SOA*, *microservices*, or *both* styles.	Yes	No
Methodical	Contains five categories that either determine the used research method (e.g. *literature study*) or the study's contribution (e.g. *model or taxonomy*).	No	Yes
Thematic	Contains nine categories that determine the topic of a publication, i.e. subfields of maintainability assurance.	Yes	Yes (except for *other*)

category had to be chosen per publication. Second, the optional *methodical* category type determined either the applied methodology (e.g. *case study*) or the provided contribution (e.g. *process or method*). It consisted of five different categories, from which multiple ones could be selected per study. Lastly, the most important *thematic* category type determined the actual maintainability-related topic of a publication, i.e. a more specific subfield of maintainability assurance. To avoid a large number of very fine-grained thematic categories, we created the generic *Other* category. At least one thematic category had to be chosen per publication. The following listing briefly describes all methodical categories.

- **Case, Field, or Empirical Study:** the publication either describes a case study (e.g. demonstrating an approach with an example system), a field study (e.g. analyzing an industry system), or an empirical study (e.g. a survey, interviews, or a controlled experiment)
- **Literature Study:** the publication presents the results of a literature study like an SLR or a systematic mapping study
- **Model or Taxonomy:** the publication contributes a (meta) model or taxonomy to further the conceptual understanding of a topic
- **Process or Method:** the publication defines a method or process, i.e. a sequence of activities to achieve a certain goal
- **Reference Architecture or Tool:** the publication either describes a reference architecture (an abstract and reusable template to create system architectures) or a tool to e.g. mitigate manual efforts

Lastly, the next listing presents all thematic categories. They refer exclusively to a service-based context.

- **Architecture Recovery and Documentation:** relying on architecture reconstruction (if no current documentation is available) or on general architecture documentation support to increase analyzability and therefore maintainability; example: [40]
- **Model-Driven Approaches:** approaches that rely on model-driven engineering to reduce long-term maintenance efforts with e.g. code generation from machine-readable models; example: [49]
- **Patterns:** applying patterns specifically designed for service orientation to systematically improve maintainability-related aspects or to describe service evolution; example: [83]
- **Antipatterns and Bad Smells:** conceptualizing service-based antipatterns and bad smells or providing detection approaches for them to identify maintainability weaknesses; example: [58]
- **Service Identification and Decomposition:** approaches to identify suitable service boundaries for functionality or to decompose large existing services into more fine-grained ones that are more beneficial for maintainability; example: [45]
- **Maintainability Metrics and Prediction:** conceptualizing or evaluating service-based metrics to analyze or predict maintainability; example: [50]
- **Change Impact and Scenarios:** approaches for analyzing the potential propagation of service changes or general scenario-based maintainability evaluation; example: [34]
- **Evolution Management:** general approaches to support or improve the overall service evolution process via e.g. systematic planning techniques, accelerating the process, increasing fault tolerance, or mitigating other negative consequences; example: [28]
- **Other:** all papers that could not be assigned to one of the other categories were sorted into this one; example: [84]

4.2 Category Distributions (RQ2)

The analysis of distributions among *architectural* categories (SOA, microservices, both) immediately revealed that nearly 90% of the 223 publications exclusively targeted SOA (see Fig. 7). Only 12 publications solely referred to microservices while an additional 12 included both SOA and microservices. This means that only a combined ∼11% of our primary studies were about maintainability assurance approaches for microservices. Since microservices are much younger than SOA, it also makes sense to look at a more recent subset, e.g. starting from 2014 when microservices began to rise in popularity (see Fig. 8). However, from 2014 to 2018, identified microservices-related publications still only accounted for a combined ∼25%.

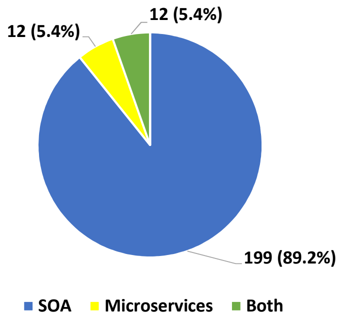


Figure 7 Architectural Categories 2007-2018.

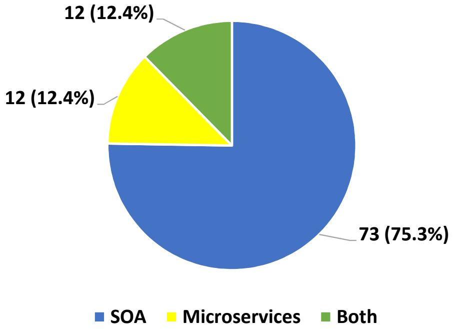


Figure 8 Architectural Categories 2014-2018.

Concerning *methodical* categories (see Fig. 9), the most frequent one was *case, field, or empirical study*: 63% of publications included such a study, most often to demonstrate or evaluate a proposed approach or to analyze industry practices. Moreover, nearly half of the publications (110) described a *process or method* as part of their contribution, which highlights the importance of systematic approaches in this field. Both the conceptual contribution of a *model or taxonomy* (75 of 223) and the more practical contribution of a *reference architecture or tool* (72 of 223) were present in roughly one third of studies. Lastly, 25 publications (11%) described a *literature study* for meta analysis. Overall, 124 publications were associated with at least two methodical categories (56%) and 68 publications with at least three (30%).

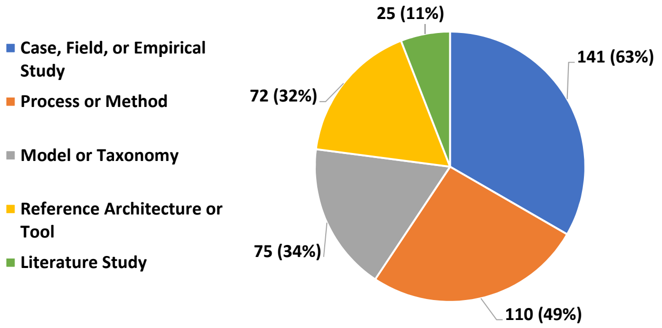


Figure 9 Distribution: Methodical Categories (percentages are relative to 223 publications).

With respect to *thematic categories* (Fig. 10), around half of the publications were equally distributed among either *evolution management* (56) or *maintainability metrics & prediction* (55). This shows the popularity of approaches to systematically manage service evolution and mitigate potential consequences as well as the interest in maintainability metrics specifically designed for service orientation. In the remaining half, the largest category accounting for 17% of total publications was *change impact & scenarios*, which indicates that qualitative evaluation has not been as popular as quantitative metric-based evaluation so far. Smaller categories were *antipatterns & bad smells* (9%), *service identification & decomposition* (9%), *patterns* (6%), *model-driven approaches* (5%), and *architecture recovery & documentation* (4%). Lastly, 18 publications were categorized with *other* (8%) because they did not fit into any existing category. As opposed to *methodical* categories, multiple selection was quite rare here, i.e. only 21 publications were related to more than one category (9%).

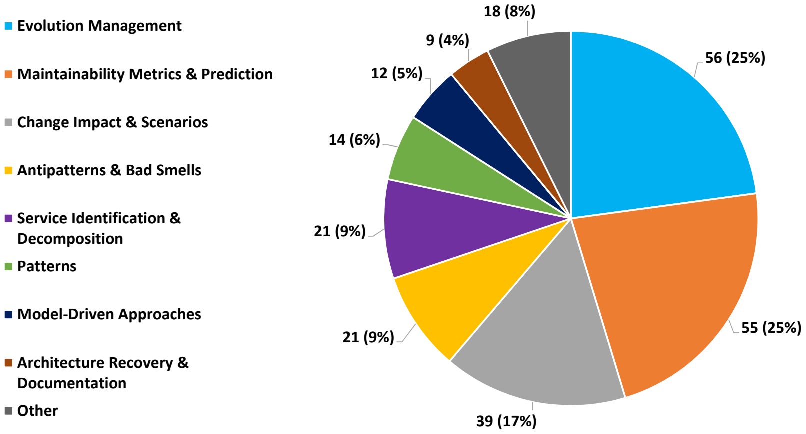


Figure 10 Distribution: Thematic Categories (percentages are relative to 223 publications).

4.3 Research Directions per Category(RQ3)

In this section, we briefly describe the most relevant research directions per identified thematic category (except for *other*). We illustrate these by describing selected exemplary studies.

Architecture Recovery and Documentation. In our smallest thematic category, the majority of the nine publications was concerned with (semi-)automatic architecture reconstruction via static analysis, dynamic runtime analysis, or a mixture of both. A static example for SOA is the intelligent search support by Reichherzer et al. [65]. Their SOAMiner tool parses and analyzes common SOA artifacts like WSDL or BPEL files and conceptualizes knowledge relevant for architecture and maintenance from them. Similarly, Buchgeher et al. [15] provide a platform to provide up-to-date architectural information of large-scale service-based systems via extraction techniques using source code and other development artifacts like POM files. With five publications, this category was also especially popular for microservices. One example is the MicroART approach of Granchelli et al. [32] that combines static information from a code repository (e.g. Docker files) with dynamic runtime data collected via `tcpdump`. A second mixed approach is the MICROLYZE framework from Kleehaus et al. [40]. By combining data from service registries and OpenTracing monitoring with static build-time information, the system's architecture can be continuously reconstructed while also taking infrastructure and hardware into account.

Overall, approaches in this category were concerned with providing accurate architecture documentation to increase analyzability and to ease maintenance efforts. Automation is used to reduce manual efforts, but this is challenging in heterogeneous and decentralized environments that consist of a very large number of diverse (micro)services. Most modern approaches combine static with dynamic analysis and sometimes even rely on tool-supported manual steps to increase accuracy.

Model-Driven Approaches. All 12 publications in this category were related to SOA and most of them were concerned with model-based verification during evolution. To support model evolution and change propagation across different model types such as business process or service models, Sindhgatta and Sengupta [71] proposed a framework that automatically analyzes Meta-Object Facility (MOF) compliant models and supports the selective application of changes to downstream models. Similarly, Liu et al. [48] designed a verification approach that is based on colored reflective Petri nets and simulates adaptive service evolution. A last approach in this area was created by Zhou et al. [88]: they analyze model consistency using hierarchical timed automata and also support the modelling of time constraints as well as architectural decomposition. In the area of business process management, Boukhebouze et al. [13] relied on rule-based specifications with the event-condition-action model to assess a business process's flexibility and to estimate its cost of change. Rules are translated into a graph and subsequently used for analysis. Lastly, Lambers et al. [44] proposed a very early holistic approach for model-driven development of service-based applications. Their goal was to enable expert users to iteratively and rapidly develop flexible services through a series of models, code generators, and a graphical user interface for visual model creation and modification.

All in all, the presented approaches rely on different kinds of formal methods to enable faster and less error-prone development and evolution of service-based systems. Models specific to service orientation like business process models were frequently used. The predominant themes were consistency checking and verification.

Patterns. We identified 14 studies that discussed service-based design patterns, i.e. best practice solution blueprints for recurring design problems, in the area of maintenance and evolution. However, contrary to previous categories, most of them did not follow one major research direction. A few publications proposed new service-based design patterns beneficial for maintainability like the *Service Decoupler* from Athanasopoulos [4]. Others like Tragatschnig et al. [76] used patterns to describe and plan the evolution of service-based systems ("change patterns"), which should increase the efficiency and correctness of modifications. Some approaches centered around existing patterns in a system. Zdun et al. [85] provided a set of constraints and metrics for automatically assessing the pattern conformance in microservice-based system to avoid architectural drift. Demange et al. [19] designed the "Service Oriented Detection Of Patterns" (SODOP) approach to automatically detect existing service-based patterns via metric rule cards so that their design quality can be evaluated. Lastly, Palma et al. [55] analyzed the change-proneness of selected service-based patterns by studying an open source system. Using the metrics *number of changes* and *code churn*, they discovered that services with patterns needed less maintenance effort, but not with a statistically significant difference.

In general, this category was surprisingly heterogeneous with diverse pattern use cases, ranging from systematic maintainability construction to architecture conformance checking or service evolution description. Unfortunately, our SLR did not identify more holistic publications with collections and discussion of service-based patterns beneficial for maintainability, e.g. a literature study. Likewise, we only identified a single empirical study [55] that analyzed the impact of service-based design patterns on modifiability.

Antipatterns and Bad Smells. With 21 publications, this category was 50% larger than *patterns*, which could indicate that preventing suboptimal designs has been perceived as more important or as more related to maintainability than to other quality attributes. Research directions in this category were also not as scattered, but followed two main themes. The smaller group was concerned with the conceptualization of service-based antipatterns. Palma and Mohay [57] presented a classification taxonomy for 20 web service and Service Component Architecture (SCA) antipatterns and also defined metrics to specify their existence. In the area of microservices, Taibi and Lenarduzzi [75] synthesized 11 common bad smells by interviewing 72 developers. Similarly, Carrasco et al. [17] collected nine microservices architecture and migration smells by analyzing 58 sources from scientific and grey literature. The larger group in this category, however, went one step further and proposed automatic detection approaches for antipatterns. An example was the SODA (Service Oriented Detection for Antipatterns) approach from Nayrolles et al. [52], which the same authors later improved by mining execution traces [51]. For RESTful services, Palma et al. [56] proposed an approach which uses semantic as well as syntactic analysis to detect linguistic antipatterns. Lastly, Sabir et al. [69] combined both directions in their SLR that not only collected existing antipatterns, but also analyzed detection approaches.

The automatic detection of service-based antipatterns to efficiently identify design flaws was the prevalent theme in this category. Different approaches like static or dynamic analysis, machine learning, genetic programming, or combinations have been proposed. An understudied area, however, seems to be the systematic refactoring of detected antipatterns.

Service Identification and Decomposition. We identified 21 publications that focused on the activities of service identification or decomposition to increase maintainability. The majority of these (15) proposed a *process or method* to derive service candidates or to decompose existing services or interfaces in a systematic or automatic fashion. A fully automated identification approach has been proposed by Leopold et al. [45]: they relied on semantic technologies to create a ranked list of service candidates from existing business process models. Athanasopoulos et al. [5] designed a tool-supported approach to analyze WSDL specifications with cohesion metrics. Based on the results, the existing interface is progressively decomposed into more cohesive units. Similarly, Daagi et al. [18] used a framework for Formal Concept Analysis (FCA) to identify hidden relations between WSDL operations and decompose the interface into several more fine-grained ones. Because numerous approaches have been proposed, there is also a decent amount of *literature studies* in this category. Kohlborn et al. [41] conducted a structured evaluation of 30 service analysis approaches and proposed a new consolidated method to address collected short-comings. A second review by Cai et al. [16] tried to keep the literature analysis closer to the general software and service engineering process and derived common "high-value activities". Lastly, a literature review from Bani-Ismail and Baghdadi [6] identified eight common service identification challenges. The same authors [7] also conducted another review to gather proposed evaluation frameworks for service identification methods.

Since a plethora of manual or automatic approaches has been proposed in this category, it becomes difficult to differentiate between them. Some publications tried to address this with literature studies, but selecting a feasible approach for a use case may still be challenging.

Maintainability Metrics and Prediction. Our second largest category consisted of 55 publications. Since existing source code or object-oriented metrics are only of limited relevance for service-oriented systems, most publications in this category proposed maintainability metrics specifically designed for service orientation. Some researchers approached this by focusing on a single maintainability-related design property like coupling [62], complexity [63], cohesion [61], or granularity [1]. Others tried to assemble holistic metric suites, like the SOA design quality model from Shim et al. [70] or the metrics suite from Sindhgatta et al. [72]. Because most proposed metrics were of an architectural nature and therefore difficult to collect from source code, some publications also focused on metrics for specific service-based artifacts like SoaML [31] or BPMN [74] diagrams. To create an overview and to compare proposed metrics, other researchers conducted literature studies. Nik Daud and Wan Kadir [54] collected and categorized service-based metrics according to structural attributes, applied phase, or artifact. Bogner et al. [11] targeted only automatically collectable metrics and also analyzed the applicability of SOA metrics for microservices. A few publications also used metrics and various machine learning techniques to predict the future maintainability of services. Wang et al. [81] used artificial neural networks to build prediction models of several web service interface metrics. In a slightly different fashion, Kumar et al. [43] applied feature selection techniques and support vector machines to evaluate the prediction quality of object-oriented metrics for the maintainability of service-based systems.

Overall, the main theme in this category was the definition of new or adapted service-based maintainability metrics. While many metrics have been proposed, their relevance and effectiveness often remained unclear. We identified only very few evaluation studies like the one from Perepletchikov and Ryan [60]. Comparative literature studies were also rare. Furthermore, even though automatic collection was a prevalent topic and several tools like Q-ImPrESS from Koziolek et al. [42] or MAAT from Engel et al. [22] were presented, the number of publicly available tools was very low, which could hinder replication studies and industry adoption. Lastly, there seems to be much potential in crossing boundaries between e.g. SOA, microservices, WSDL, or SoaML to adapt evaluated metrics to other fields.

Change Impact and Scenarios. A focus on dependencies between clients and services has made change impact analysis a popular service-based topic. This category also comprises publications about more general scenario-based maintainability evaluation. In total, we identified 39 papers, which made this the third largest category. A large number of these publications proposes specific approaches. Wang and Capretz [82] combined information entropy with dependency analysis to quantify the relative importance of a service for change effects. Another approach is taken by Hirzalla et al. [34]: they created a framework (IntelliTrace), which relies on the modeling of traceability links between SOA artifacts like business goals, processes, or services. Based on these links, the impact of changes at different levels can be analyzed. Lastly, Khanh Dam [38] designed an approach based on association rule data mining to predict change impact using the version history of web services. To collect and compare proposed approaches, Amjad Alam et al. [2] conducted an SLR about impact analysis and change propagation for business processes and SOA. Their analysis of 43 studies concluded that very few mature approaches and tools existed, especially for bottom-up or cross-organizational analysis. The second major research direction was concerned with analysis of existing systems or APIs to derive information about their evolution change impact. Using a tool called WSDLDiff, Romano and Pinzger [67] extracted and analyzed fine-grained changes from the WSDL version histories of four web services from Amazon and FedEx. Similarly, Espinha et al. [24] analyzed the evolution of the Twitter, Google Maps, Facebook, and Netflix APIs. By interviewing six client developers and by analyzing source code version histories, they investigated how the API evolution affected service consumers.

The two major research directions in this category were a) proposing approaches for change impact analysis and b) empirical studies on the evolution impact of existing service-based systems. Proposed methods were mostly based on dependency graphs or repository data mining. Artifacts specific for service orientation like WSDL files were often used. Very few publications, however, were concerned with general scenario-based evaluation, like the one from Leotta et al. [46], who used the Software Architecture Analysis Method (SAAM) to compare the maintainability of one SOA and one non-SOA alternative. Our review did not identify a scenario-based method specifically designed for service orientation.

Evolution Management. With 56 papers, evolution management was our largest category (25%). Since it was also our most general one, it consisted of diverse approaches to control and plan service evolution. Therefore, no major research directions could be identified. However, one similarity among these publications was that most of them (48 of 56) proposed either a *process or method* or a *model or taxonomy*, i.e. most work was conceptual in nature. To illustrate this diversity, we present some selected approaches. Zhang et al. [86] designed a framework to manage requirements evolution in service-based system. The framework is based on Role, Goal, Process and Service (RGPS) elements and also contains a meta model and strategies. Another model and methodology was proposed by Zuo et al. [90]. In their change-centric model for web service evolution, they specify e.g. stakeholder behavior, service versioning, and the details of service changes. Feng et al. [25] created a taxonomy framework for SOA evolution that describes the motivation, location, time, and support mechanism of changes. The goal of their work is to support the analysis and planning of service evolution. Lastly, a more holistic framework for web service evolution support (WSDarwin) was presented by Fokaefs [27]. WSDarwin consists of an Eclipse plugin for automatic service client adaptation on interface changes, a web application to automatically generate WADL documentation for RESTful services and to compare different WADL versions, and a decision support system for evolving service ecosystems based on game theory (see also [28]).

4.4 Differences Between Approaches for SOA and Microservices (RQ4)

When analyzing differences between the identified publications for SOA and those for microservices, the most apparent finding was the small percentage of microservice-focused studies (less than 11% or less than 25% for 2014-2018). While microservices are the younger paradigm, it seems that the scientific interest in their maintainability assurance is just getting started. Possible reasons could be that most microservice-based systems are still fairly young and therefore decently maintainable or that their inherent evolution qualities are perceived as more beneficial when compared to SOA. With respect to *thematic* categories (see Fig. 11), we see three categories without approaches exclusively designed for microservices (*evolution management*, *change impact & scenarios*, and *model-driven approaches*), but only one without a single publication on microservices, namely *model-driven approaches*. While such approaches do exist for microservices [64], they were not identified by our review, maybe because they are not advertised for maintainability. The most prominent category for microservices was *architecture recovery & documentation* (5 of 9 papers, all of them for recovery), which highlights the importance of this topic. While automatic microservice decomposition and extraction is also a popular topic in academia [30], only two of the 21 papers in *service identification & decomposition* were about microservices. This is mainly due to the fact that we excluded pure legacy migration approaches. Lastly, for *antipatterns & bad smells*, publications for microservices were mostly concerned with defining antipatterns while more established SOA publications already proposed automatic detection approaches. This may be a sensible next step for microservices.

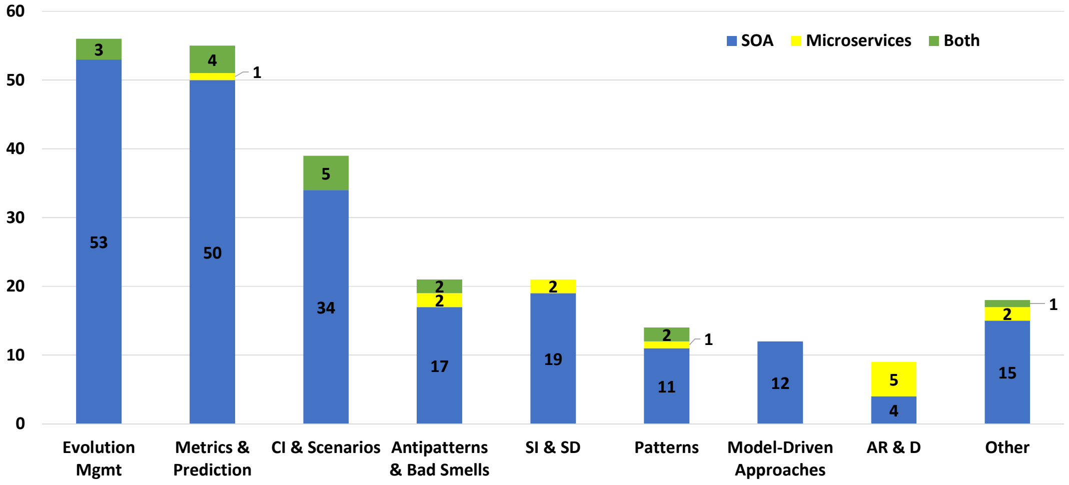


Figure 11 Distribution: Thematic Categories Grouped By Architectural Category.

In general, we identified a lot of potential for the adoption of SOA approaches for microservices, especially in the areas of *maintainability metrics & prediction*, *antipatterns & bad smells*, *service identification & decomposition*, and *patterns*. Existing SOA research in these categories could be valuable for the evolution and maintenance of microservices if the techniques are also applicable for e.g. strong decentralization or technological heterogeneity. Moreover, the majority of studies on RESTful services should be directly applicable for microservice-based systems. Lastly, our SLR did not identify studies about possible negative maintainability impacts of microservices, e.g. in the areas of knowledge exchange, team synchronization, technological heterogeneity, or code duplication. While the maintainability of microservices as an architectural style is generally perceived as positive, we still see the potential for empirical studies on these topics.

5 Threats to Validity

Results derived from systematic literature studies may suffer from limitations in different areas if not performed with great rigor (see e.g. [87]). Even though we adhered to a detailed SLR protocol and the complete study selection and categorization was performed by two researchers to mitigate subjective bias, there is still the possibility for threats to validity. One example for the planning phase is that our search strategy could have been insufficient due to missing keywords or not included databases. Likewise, the presentation of exemplary publications to highlight existing directions per category was subject to our own perception of relevance. Other researchers may disagree or come to different conclusions. However, the most prominent threat to validity with this SLR is most likely the limiting of search results to the first 250 entries per publisher (1000 papers from four publishers). This made the results dependent on the relevance sorting of each search engine and may also hinder replicability if publishers decide to change their algorithms in the future. Even though our snowballing nearly doubled the amount of selected primary studies, there is still the possibility that we may have missed relevant literature branches without links to our initial set. We accepted this threat to keep the effort manageable within the project time frame.

6 Conclusion

Since the scientific literature on maintainability assurance for service-oriented systems is diverse and scattered, we conducted an SLR with the goal to categorize the proposed approaches and to analyze differences between SOA and microservices. As an answer to RQ1, we derived a categorization set with *architectural* (SOA, microservices, both), *methodical* (method or contribution), and *thematic* (subfield of maintainability assurance) categories from the 223 selected primary studies. The distribution analysis (RQ2) revealed for example that nearly 90% of papers exclusively targeted SOA (199) and that *evolution management* and *maintainability metrics & prediction* were the most prominent thematic categories (both with ∼25%). For each thematic category, we also presented the most relevant research directions with illustrating studies (RQ3). Exemplary differences between approaches for SOA and microservices (RQ4) were the importance of architecture reconstruction and the absence of automatic antipattern detection approaches for microservices. While there was only a small number of approaches for microservices, we identified a lot of potential for adapting SOA approaches in several categories.

Future research could be concerned with literature studies for individual categories to provide more insights into these subfields and to analyze the adoption potential for microservices in greater detail. An analysis of maintainability-related differences between orchestration (SOA) and choreography (microservices) across the primary studies may also yield helpful results for the usage of these two paradigms. Lastly, it could be interesting to replicate this SLR exclusively for microservices in a few years when more publications exist for these topics. To enable replication and to allow convenient reuse of our results, we shared all SLR artifacts in a GitHub repository[2].

[2] https://github.com/xJREB/slr-maintainability-assurance

References

1 Saad Alahmari, Ed Zaluska, and David C. De Roure. A Metrics Framework for Evaluating SOA Service Granularity. In *2011 IEEE International Conference on Services Computing*, pages 512–519. IEEE, July 2011. doi:10.1109/SCC.2011.98.

2 Khubaib Amjad Alam, Rodina Binti Ahmad, and Maria Akhtar. Change Impact analysis and propagation in service based business process management systems preliminary results from a systematic review. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 7–12. IEEE, September 2014. doi:10.1109/MySec.2014.6985981.

3 Marc Andreessen. Why Software Is Eating The World. *Wall Street Journal*, 20, 2011. URL: https://www.wsj.com/articles/SB10001424053111903480904576512250915629460.

4 Dionysis Athanasopoulos. Service Decoupler: Full Dynamic Decoupling in Service Invocation. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs - EuroPLoP '17*, pages 1–9, New York, New York, USA, 2017. ACM Press. doi:10.1145/3147704.3147716.

5 Dionysis Athanasopoulos, Apostolos V. Zarras, George Miskos, Valerie Issarny, and Panos Vassiliadis. Cohesion-Driven Decomposition of Service Interfaces without Access to Source Code. *IEEE Transactions on Services Computing*, 8(4):550–5532, 2015. doi:10.1109/TSC.2014.2310195.

6 Basel Bani-Ismail and Youcef Baghdadi. A Literature Review on Service Identification Challenges in Service Oriented Architecture. In *Communications in Computer and Information Science*, pages 203–214. Springer, Cham, 2018. doi:10.1007/978-3-319-95204-8_18.

7 Basel Bani-Ismail and Youcef Baghdadi. A Survey of Existing Evaluation Frameworks for Service Identification Methods: Towards a Comprehensive Evaluation Framework. In *Communications in Computer and Information Science*, volume 877, pages 191–202. Springer, Cham, August 2018. doi:10.1007/978-3-319-95204-8_17.

8 Justus Bogner, Tobias Boceck, Matthias Popp, Dennis Tschechlov, Stefan Wagner, and Alfred Zimmermann. Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 95–101, Hamburg, Germany, March 2019. IEEE. doi:10.1109/ICSA-C.2019.00025.

9 Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Limiting Technical Debt with Maintainability Assurance – An Industry Survey on Used Techniques and Differences with Service- and Microservice-Based Systems. In *Proceedings of the 2018 International Conference on Technical Debt - TechDebt '18*, pages 125–133, New York, New York, USA, 2018. ACM Press. doi:10.1145/3194164.3194166.

10 Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, Ohio, USA, 2019. IEEE.

11 Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Automatically measuring the maintainability of service- and microservice-based systems. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement on - IWSM Mensura '17*, pages 107–115, New York, New York, USA, 2017. ACM Press. doi:10.1145/3143434.3143443.

12 Justus Bogner, Alfred Zimmermann, and Stefan Wagner. Analyzing the Relevance of SOA Patterns for Microservice-Based Systems. In *Proceedings of the 10th Central European Workshop on Services and their Composition (ZEUS'18)*, pages 9–16, Dresden, Germany, 2018. CEUR-WS.org.

13 Mohamed Boukhebouze, Youssef Amghar, Aïcha Nabila Benharkat, and Zakaria Maamar. A rule-based approach to model and verify flexible business processes. *International Journal of Business Process Integration and Management*, 5(4):287, 2011. doi:10.1504/IJBPIM.2011.043389.

14 Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012. doi:10.1016/j.infsof.2011.06.002.

15 Georg Buchgeher, Rainer Weinreich, and Heinz Huber. A Platform for the Automated Provisioning of Architecture Information for Large-Scale Service-Oriented Software Systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11048 LNCS, pages 203–218. Springer International Publishing, 2018. doi:10.1007/978-3-030-00761-4_14.

16 Simin Cai, Yan Liu, and Xiaoping Wang. A Survey of Service Identification Strategies. In *2011 IEEE Asia-Pacific Services Computing Conference*, pages 464–470. IEEE, December 2011. doi:10.1109/APSCC.2011.12.

17 Andrés Carrasco, Brent van Bladel, and Serge Demeyer. Migrating towards microservices: migration and architecture smells. In *Proceedings of the 2nd International Workshop on Refactoring - IWoR 2018*, pages 1–6, New York, New York, USA, 2018. ACM Press. doi:10.1145/3242163.3242164.

18 Marwa Daagi, Ali Ouniy, Marouane Kessentini, Mohamed Mohsen Gammoudi, and Salah Bouktif. Web Service Interface Decomposition Using Formal Concept Analysis. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 172–179. IEEE, June 2017. doi:10.1109/ICWS.2017.30.

19 Anthony Demange, Naouel Moha, and Guy Tremblay. Detection of SOA Patterns. In *Service-Oriented Computing. ICSOC 2013. Lecture Notes in Computer Science*, pages 114–130. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-45005-1_9.

20 Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150(April):77–97, April 2019. doi:10.1016/j.jss.2019.01.001.

21 Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-67425-4_12.

22 Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In Jan Mendling and Haralambos Mouratidis, editors, *Lecture Notes in Business Information Processing*, volume 317 of *Lecture Notes in Business Information Processing*, pages 74–89. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-92901-9_8.

23 Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

24 Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, February 2015. doi:10.1016/j.jss.2014.10.014.

25 Zaiwen Feng, Patrick C. K. Hung, Keqing He, Yutao Ma, Matthias Farwick, Bing Li, and Rong Peng. Towards a Taxonomy Framework of Evolution for SOA Solution: From a Practical Point of View. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7652 LNCS, pages 261–274. Springer, Berlin, Heidelberg, October 2013. doi:10.1007/978-3-642-38333-5_28.

26 José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software & Systems Modeling*, 12(2):349–367, May 2013. doi:10.1007/s10270-012-0236-1.

27 Marios Fokaefs. WSDarwin: A Framework for the Support of Web Service Evolution. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 668–668. IEEE, September 2014. doi:10.1109/ICSME.2014.123.

28 Marios Fokaefs and Eleni Stroulia. Software Evolution in Web-Service Ecosystems: A Game-Theoretic Model. *Service Science*, 8(1):1–18, March 2016. doi:10.1287/serv.2015.0114.

29 Martin Fowler. Microservices Resource Guide, 2015. URL: `http://martinfowler.com/microservices`.

30 Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From Monolith to Microservices: A Classification of Refactoring Approaches. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 128–141. Springer, Toulouse, France, 2019. `doi:10.1007/978-3-030-06019-0_10`.

31 Michael Gebhart, Marc Baumgartner, Stephan Oehlert, Martin Blersch, and Sebastian Abeck. Evaluation of Service Designs Based on SoaML. In *2010 Fifth International Conference on Software Engineering Advances*, pages 7–13. IEEE, August 2010. `doi:10.1109/ICSEA.2010.8`.

32 Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards Recovering the Software Architecture of Microservice-Based Systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53. IEEE, April 2017. `doi:10.1109/ICSAW.2017.48`.

33 Qing Gu and Patricia Lago. Exploring service-oriented system engineering challenges: a systematic literature review. *Service Oriented Computing and Applications*, 3(3):171–188, September 2009. `doi:10.1007/s11761-009-0046-7`.

34 M.A. Hirzalla, A. Zisman, and J. Cleland-Huang. Using Traceability to Support SOA Impact Analysis. In *2011 IEEE World Congress on Services*, pages 145–152. IEEE, July 2011. `doi:10.1109/SERVICES.2011.103`.

35 International Organization For Standardization. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

36 Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, page 504, New York, New York, USA, 2012. ACM Press. `doi:10.1145/2245276.2245373`.

37 Miguel A. Jimenez, Angela Villota Gomez, Norha M. Villegas, Gabriel Tamura, and Laurence Duchien. A Framework for Automated and Composable Testing of Component-Based Services. In *2014 IEEE 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10. IEEE, September 2014. `doi:10.1109/MESOCA.2014.9`.

38 Hoa Khanh Dam. Predicting change impact in Web service ecosystems. *International Journal of Web Information Systems*, 10(3):275–290, August 2014. `doi:10.1108/IJWIS-03-2014-0006`.

39 Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature reviews in Software Engineering. Technical report, School of Computer Science and Mathematics, Keele University, Keele, UK, 2007.

40 Martin Kleehaus, Ömer Uludağ, Patrick Schäfer, and Florian Matthes. MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-Based Environments. In *Lecture Notes in Business Information Processing*, volume 317, pages 148–162. Springer, Cham, June 2018. `doi:10.1007/978-3-319-92901-9_14`.

41 Thomas Kohlborn, Axel Korthaus, Taizan Chan, and Michael Rosemann. Identification and Analysis of Business and Software Services—A Consolidated Approach. *IEEE Transactions on Services Computing*, 2(1):50–64, January 2009. `doi:10.1109/TSC.2009.6`.

42 Heiko Koziolek, Bastian Schlich, Carlos Bilich, Roland Weiss, Steffen Becker, Klaus Krogmann, Mircea Trifu, Raffaela Mirandola, and Anne Koziolek. An industrial case study on quality impact prediction for evolving service-oriented software. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 776, New York, New York, USA, 2011. ACM Press. `doi:10.1145/1985793.1985902`.

43 Lov Kumar, Aneesh Krishna, and Santanu Ku. Rath. The impact of feature selection on maintainability prediction of service-oriented applications. *Service Oriented Computing and Applications*, 11(2):137–161, June 2017. `doi:10.1007/s11761-016-0202-9`.

44 Leen Lambers, Hartmut Ehrig, Leonardo Mariani, and Mauro Pezzè. Iterative model-driven development of adaptable service-based applications. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 453, New York, New York, USA, 2007. ACM Press. doi:10.1145/1321631.1321707.

45 Henrik Leopold, Fabian Pittke, and Jan Mendling. Automatic service derivation from business process model repositories via semantic technology. *Journal of Systems and Software*, 108:134–147, October 2015. doi:10.1016/j.jss.2015.06.007.

46 Maurizio Leotta, Filippo Ricca, Gianna Reggio, and Egidio Astesiano. Comparing the Maintainability of Two Alternative Architectures of a Postal System: SOA vs. Non-SOA. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 317–320. IEEE, March 2011. doi:10.1109/CSMR.2011.41.

47 Grace Lewis and Dennis B. Smith. Developing Realistic Approaches for the Migration of Legacy Components to Service-Oriented Architecture Environments. In *Trends in Enterprise Application Architecture*, pages 226–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-75912-6_17.

48 Ying Liu, Walter Cazzola, and Bin Zhang. Towards a colored reflective Petri-net approach to model self-evolving service-oriented architectures. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, page 1858, New York, New York, USA, 2012. ACM Press. doi:10.1145/2245276.2232081.

49 Christine Mayr, Uwe Zdun, and Schahram Dustdar. View-based model-driven architecture for enhancing maintainability of data access services. *Data & Knowledge Engineering*, 70(9):794–819, September 2011. doi:10.1016/j.datak.2011.05.004.

50 Arafat Abdulgader Mohammed Elhag and Radziah Mohamad. Metrics for evaluating the quality of service-oriented design. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 154–159. IEEE, September 2014. doi:10.1109/MySec.2014.6986006.

51 Mathieu Nayrolles, Naouel Moha, and Petko Valtchev. Improving SOA antipatterns detection in Service Based Systems by mining execution traces. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 321–330. IEEE, October 2013. doi:10.1109/WCRE.2013.6671307.

52 Mathieu Nayrolles, Francis Palma, Naouel Moha, and Yann-Gaël Guéhéneuc. Soda: A Tool Support for the Detection of SOA Antipatterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7759 LNCS, pages 451–455. Springer, Berlin, Heidelberg, November 2013. doi:10.1007/978-3-642-37804-1_51.

53 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition, 2015.

54 Nik Marsyahariani Nik Daud and Wan M. N. Wan Kadir. Static and dynamic classifications for SOA structural attributes metrics. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*, pages 130–135. IEEE, September 2014. doi:10.1109/MySec.2014.6986002.

55 Francis Palma, Le An, Foutse Khomh, Naouel Moha, and Yann-Gael Gueheneuc. Investigating the Change-Proneness of Service Patterns and Antipatterns. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 1–8. IEEE, November 2014. doi:10.1109/SOCA.2014.43.

56 Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Guy Tremblay. Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9435, pages 171–187. Springer, Berlin, Heidelberg, 2015. doi:10.1007/978-3-662-48616-0_11.

57 Francis Palma and Naouel Mohay. A study on the taxonomy of service antipatterns. In *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, pages 5–8. IEEE, March 2015. doi:10.1109/PPAP.2015.7076848.

58 Francis Palma, Mathieu Nayrolles, Naouel Moha, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel. SOA antipatterns: an approach for their specification and detection.

International Journal of Cooperative Information Systems, 22(04):1341004, December 2013. doi:10.1142/S0218843013410049.

59 M.P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE'03)*. IEEE Comput. Soc, 2003. doi:10.1109/WISE.2003.1254461.

60 Mikhail Perepletchikov and Caspar Ryan. A Controlled Experiment for Evaluating the Impact of Coupling on the Maintainability of Service-Oriented Software. *IEEE Transactions on Software Engineering*, 37(4):449–465, July 2011. doi:10.1109/TSE.2010.61.

61 Mikhail Perepletchikov, Caspar Ryan, and Keith Frampton. Cohesion Metrics for Predicting Maintainability of Service-Oriented Software. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 328–335. IEEE, 2007. doi:10.1109/QSIC.2007.4385516.

62 Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. In *2007 Australian Software Engineering Conference (ASWEC'07)*, pages 329–340. IEEE, April 2007. doi:10.1109/ASWEC.2007.17.

63 Zhang Qingqing and Li Xinke. Complexity Metrics for Service-Oriented Systems. In *2009 Second International Symposium on Knowledge Acquisition and Modeling*, volume 3, pages 375–378. IEEE, 2009. doi:10.1109/KAM.2009.90.

64 Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. A model-driven workflow for distributed microservice development. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing - SAC '19*, pages 1260–1262, New York, New York, USA, 2019. ACM Press. doi:10.1145/3297280.3300182.

65 Thomas Reichherzer, Eman El-Sheikh, Norman Wilde, Laura White, John Coffey, and Sharon Simmons. Towards intelligent search support for web services evolution identifying the right abstractions. In *2011 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 53–58. IEEE, September 2011. doi:10.1109/WSE.2011.6081819.

66 Mark Richards. *Microservices vs. Service-Oriented Architecture.* O'Reilly Media, Sebastopol, CA, 2016.

67 Daniele Romano and Martin Pinzger. Analyzing the Evolution of Web Services Using Fine-Grained Changes. In *2012 IEEE 19th International Conference on Web Services*, pages 392–399. IEEE, June 2012. doi:10.1109/ICWS.2012.29.

68 David Rowe, John Leaney, and David Lowe. Defining systems architecture evolvability - a taxonomy of change. In *International Conference on the Engineering of Computer-Based Systems*, pages 45–52. IEEE, 1998. doi:10.1109/ECBS.1998.10027.

69 Fatima Sabir, Francis Palma, Ghulam Rasool, Yann-Gaël Guéhéneuc, and Naouel Moha. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, 49(1):3–39, January 2019. doi:10.1002/spe.2639.

70 Bingu Shim, Siho Choue, Suntae Kim, and Sooyong Park. A Design Quality Model for Service-Oriented Architecture. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 403–410. IEEE, 2008. doi:10.1109/APSEC.2008.32.

71 Renuka Sindhgatta and Bikram Sengupta. An extensible framework for tracing model evolution in SOA solution design. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, page 647, New York, New York, USA, 2009. ACM Press. doi:10.1145/1639950.1639960.

72 Renuka Sindhgatta, Bikram Sengupta, and Karthikeyan Ponnalagu. Measuring the Quality of Service Oriented Design. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5900 LNCS, pages 485–499. Springer, Berlin, Heidelberg, November 2009. doi:10.1007/978-3-642-10383-4_36.

73 Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146(September):215–232, December 2018. doi:10.1016/j.jss.2018.09.082.

74 Iis Solichah, Margaret Hamilton, Petrus Mursanto, Caspar Ryan, and Mikhail Perepletchikov. Exploration on software complexity metrics for business process model and notation. In *2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 31–37. IEEE, September 2013. doi:10.1109/ICACSIS.2013.6761549.

75 Davide Taibi and Valentina Lenarduzzi. On the Definition of Microservice Bad Smells. *IEEE Software*, 35(3):56–62, May 2018. doi:10.1109/MS.2018.2141031.

76 Simon Tragatschnig, Srdjan Stevanetic, and Uwe Zdun. Supporting the evolution of event-driven service-oriented architectures using change patterns. *Information and Software Technology*, 100:133–146, August 2018. doi:10.1016/j.infsof.2018.04.005.

77 Colin C. Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, 138(December):174–188, April 2018. doi:10.1016/j.jss.2017.12.026.

78 Dirk Voelz and Andreas Goeb. What is Different in Quality Management for SOA? In *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 47–56. IEEE, October 2010. doi:10.1109/EDOC.2010.27.

79 Stefan Wagner. *Software Product Quality Control*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38571-1.

80 Allen Wang and Sudhir Tonse. Announcing Ribbon: Tying the Netflix Mid-Tier Services Together, 2013. URL: https://medium.com/netflix-techblog/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62.

81 Hanzhang Wang, Marouane Kessentini, and Ali Ouni. Prediction of Web Services Evolution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9936 LNCS, pages 282–297. Springer, Cham, October 2016. doi:10.1007/978-3-319-46295-0_18.

82 Shuying Wang and Miriam A.M. Capretz. Dependency and Entropy Based Impact Analysis for Service-Oriented System Evolution. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 412–417. IEEE, August 2011. doi:10.1109/WI-IAT.2011.196.

83 Shuying Wang, Wilson Akio Higashino, Michael Hayes, and Miriam A M Capretz. Service Evolution Patterns. In *2014 IEEE International Conference on Web Services*, pages 201–208. IEEE, June 2014. doi:10.1109/ICWS.2014.39.

84 Laura White, Norman Wilde, Thomas Reichherzer, Eman El-Sheikh, George Goehring, Arthur Baskin, Ben Hartmann, and Mircea Manea. Understanding Interoperable Systems: Challenges for the Maintenance of SOA Applications. In *2012 45th Hawaii International Conference on System Sciences*, pages 2199–2206. IEEE, January 2012. doi:10.1109/HICSS.2012.614.

85 Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In *Service-Oriented Computing*, volume 10601, pages 411–429, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-69035-3_29.

86 Songlin Zhang, Junsong Yin, and Rong Liu. A RGPS-based framework for service-oriented requirement evolution of networked software. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 321–325. IEEE, May 2011. doi:10.1109/ICCSN.2011.6013724.

87 Xin Zhou, Yuqin Jin, He Zhang, Shanshan Li, and Xin Huang. A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160, Hamilton, New Zealand, 2016. IEEE. doi:10.1109/APSEC.2016.031.

88 Yu Zhou, Jidong Ge, Pengcheng Zhang, and Weigang Wu. Model based verification of dynamically evolvable service oriented systems. *Science China Information Sciences*, 59(3):32101, March 2016. doi:10.1007/s11432-015-5332-8.

89 Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3-4):301–310, July 2017. doi:10.1007/s00450-016-0337-0.

90 Wei Zuo, Aicha Nabila Benharkat, and Youssef Amghar. Change-centric Model for Web Service Evolution. In *2014 IEEE International Conference on Web Services*, pages 712–713. IEEE, June 2014. doi:10.1109/ICWS.2014.111.

Introduction to Microservice API Patterns (MAP)

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland
ozimmerm@hsr.ch

Mirko Stocker
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland
mirko.stocker@hsr.ch

Daniel Lübke
iQuest GmbH, Hanover, Germany
ich@daniel-luebke.de

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland
c.pautasso@ieee.org

Uwe Zdun
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria
uwe.zdun@univie.ac.at

Abstract

The Microservice API Patterns (MAP) language and supporting website premiered under this name at Microservices 2019. MAP distills proven, platform- and technology-independent solutions to recurring (micro-)service design and interface specification problems such as finding well-fitting service granularities, rightsizing message representations, and managing the evolution of APIs and their implementations. In this paper, we motivate the need for such a pattern language, outline the language organization and present two exemplary patterns describing alternative options for representing nested data. We also identify future research and development directions.

2012 ACM Subject Classification Software and its engineering → Patterns; Software and its engineering → Designing software

Keywords and phrases application programming interfaces, distributed systems, enterprise application integration, service-oriented computing, software architecture

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.4

1 Motivation

It is hard to escape the term *microservices* these days. Much has been said about this rather advanced approach to system decomposition since its inception a few years ago [10]. The basic elements of a microservice-based message exchange are introduced in Figure 1.

Early adopters' experiences suggest that service design requires particular attention if microservices are supposed to deliver on their promises [16]:

- How many (micro-)service operations should be exposed in Application Programming Interfaces (APIs)?
- Which service cuts let services and their clients deliver user value jointly, but couple them loosely?
- How often do services and their clients interact to exchange data? How much and which data should be exchanged?
- What are suitable message representation structures and nesting levels, and how do these change throughout service life cycles?

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 4; pp. 4:1–4:17
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

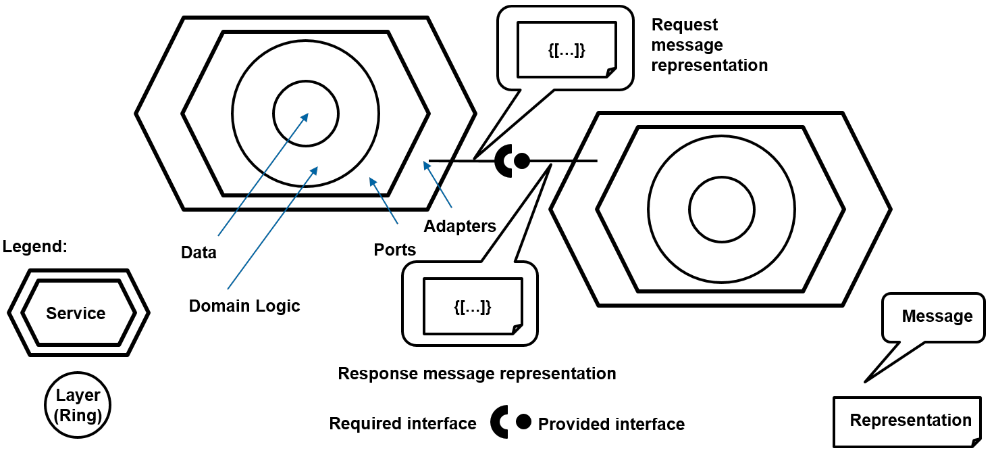


Figure 1 Microservices, represented as hexagons, exchange request and response message representations via platform-independent *ports* and technology-specific *adapters*. The inner structure of the services is sketched in onion form: each ring represents a local logical layer (e.g., *logic*, *data*).

- How can the meaning of message representations be agreed upon – and how to stick to these contracts in the long run?

To address these and related design issues and choose working combinations out of the many possible design options, application context and requirements must be analyzed. Our Microservice API Patterns (MAP) cover and organize this design space. Before we describe MAP and present two example patterns in the following sections, let us first recapitulate what microservices actually are (and where they came from).

1.1 A Consolidated Definition of Microservices

Microservices architectures have evolved from previous incarnations of Service-Oriented Architectures (SOAs) [5]. They consist of independently deployable, scalable and changeable services, each having a single responsibility. These responsibilities model business capabilities. Microservices often are deployed in lightweight virtualization containers, encapsulate their own state, and communicate via message-based remote APIs in a loosely coupled fashion. Microservices solutions leverage polyglot programming, polyglot persistence, as well as DevOps practices including decentralized continuous delivery and end-to-end monitoring [22], [13], [10].

When it comes to protocol selection, message-based APIs such as RESTful HTTP or queue-based event sourcing and streaming have come to dominate over remote procedure calls, including their object-oriented variants [15]. JSON is a particularly popular data serialization and message exchange format in many developer communities today.

1.2 Service Design Challenges

Microservices architectures include many remote APIs. The data representations exposed by these APIs must not only meet the information and processing needs of clients and other services, but also be designed and documented in an understandable and maintainable way.

While microservice API design and implementation might seem to be simple and straightforward from the distance, a closer look unveils that a lot of interesting problems are awaiting API teams:

- *Requirements diversity*: The wants and needs of API clients differ from one another, and keep on changing. API providers have to decide whether they want to offer good-enough compromises in a single unified API or try to satisfy all client requirements individually.
- *Design mismatches*: What backend systems can do (in terms of functional scope and quality), and how they are structured (in terms of endpoint and data definitions), might be different from what clients expect. These differences have to be overcome.
- *Open vs. closed systems*: API clients and providers often have conflicting goals. For instance, the desire to innovate and market dynamics such as competing API providers trying to catch up on each other may cause more change and possibly incompatible evolution strategies than clients are able or willing to accept. Publishing an API means opening up a system and giving up some control, thus limiting the freedom to change it. Clients might use data that is exposed by an API in unexpected ways.
- *Stability vs. flexibility*: Microservices help to enable frequent releases, e.g., in the context of DevOps practices such as continuous delivery. Changes are released at an ever increasing pace. In contrast, APIs should stay as stable as possible to avoid breaking client code. This constant conflict needs to be resolved by microservice API designers.

These conflicting requirements and stakeholder concerns must be balanced; many design trade-offs can be observed:

- *Few operations that carry lots of data back and forth vs. many chatty, fine-grained interactions.* Which choice is better in terms of performance, scalability, bandwidth consumption and evolvability?
- *Stable, standardized, elaborate interfaces vs. fast changing, specialized, focused ones.* How to find a balance between breadth and depth? How to keep the interfaces compatible without sacrificing their extensibility?
- *Data consistency vs. reliability and fast response times.* Should state changes be reported via coordinated API calls or via reactive event sourcing and streaming? Should commands and queries be separated architecturally? To which extent can and should consistency, availability, and recoverability (backup) requirements be satisfied? [14]

1.3 Existing Design Heuristics

One can find many excellent books providing deep advice about using RESTful HTTP, e.g., which HTTP verb or method to pick to implement a particular operation, or how to apply asynchronous messaging including routing, transformation, and guaranteed delivery [1], [7]. Strategic Domain-Driven Design [3], [19] can assist with service identification. SOA, cloud and microservice infrastructure patterns have already been proposed, and structuring data storages also is understood well. Our previous publications [17] and [23] cover such related works; the MAP website also gives reading recommendations[1].

Structuring data exchanges without breaking information hiding remains hard; no single solution exists. According to Helland [4], “data on the outside” differs from “data on the inside” significantly. Data access/usage profiles drive many data modeling decisions, both for data on the inside and for data in the outside. However, inside and outside data have diverging mutability, lifetime, accuracy, consistency and protection needs.

[1] `https://microservice-api-patterns.org/relatedPatternLanguages`

2 Microservice API Patterns (MAP) Scope and Organization

Microservice API Patterns (MAP) takes a broad view on microservice API design and evolution, from the perspective of data on the outside, i.e., the message representations and payloads exchanged when APIs are called (as shown in Figure 1). These messages are structured as *representation elements* which differ in their meaning and structure as API endpoints and their operations have different architectural roles and responsibilities. Critical design choices about the message structure and semantics strongly influence the design time and runtime qualities of an API and its underlying microservices implementations. Many options exist, with very different characteristics. The API designs evolve over time.

2.1 Patterns as Knowledge Sharing Vehicles

Software patterns are proven knowledge sharing vehicles with a 25-year track record [6]. We decided for the pattern format to share API design advice because:

- Pattern names aim at forming a domain vocabulary, a *ubiquitous language* [3]. For instance, Hohpe's and Woolf's Enterprise Integration Patterns [7] have become the lingua franca of queue-based messaging; they are implemented in a number of frameworks and tools. Such ubiquitous language for API design is missing to date.
- The forces and consequences sections of patterns support informed decision making, for instance about desired and achievable quality characteristics (but also downsides of certain designs). The design challenges and trade-offs identified in Section 1 frame and support such design discussions.
- Patterns are soft around their edges: they only sketch solutions and do not provide blueprints to be followed blindly.
- Patterns are not invented, but mined from practical experience and then curated and hardened via peer feedback.

2.2 Knowledge Categories

MAP addresses the following questions, which also define pattern categories:

- The *structure* of messages and the message elements that play critical roles in the design of APIs. What is an adequate number of representation elements for request and response messages? How are these elements structured? How can they be grouped and annotated with supplemental usage information (metadata)? [23]
- The impact of message content on the *quality* of the API. How can an API provider achieve a certain level of quality of the offered API, while at the same time using its available resources in a cost-effective way? How can the quality trade-offs be communicated and accounted for? [17]
- The roles and *responsibilities* [20] of API operations. Which is the architectural role played by each API endpoint and its operations? How do these roles and their responsibilities impact microservice size and granularity?
- API descriptions as a means for API governance and *evolution* over time. How to deal with life cycle management concerns such as support periods and versioning? How to promote backward compatibility and extensibility? How to communicate breaking changes? [11]

Two more categories complete the language scope, *foundation* and *identification* (not covered here due to space constraints). See Figure 2 for an overview.

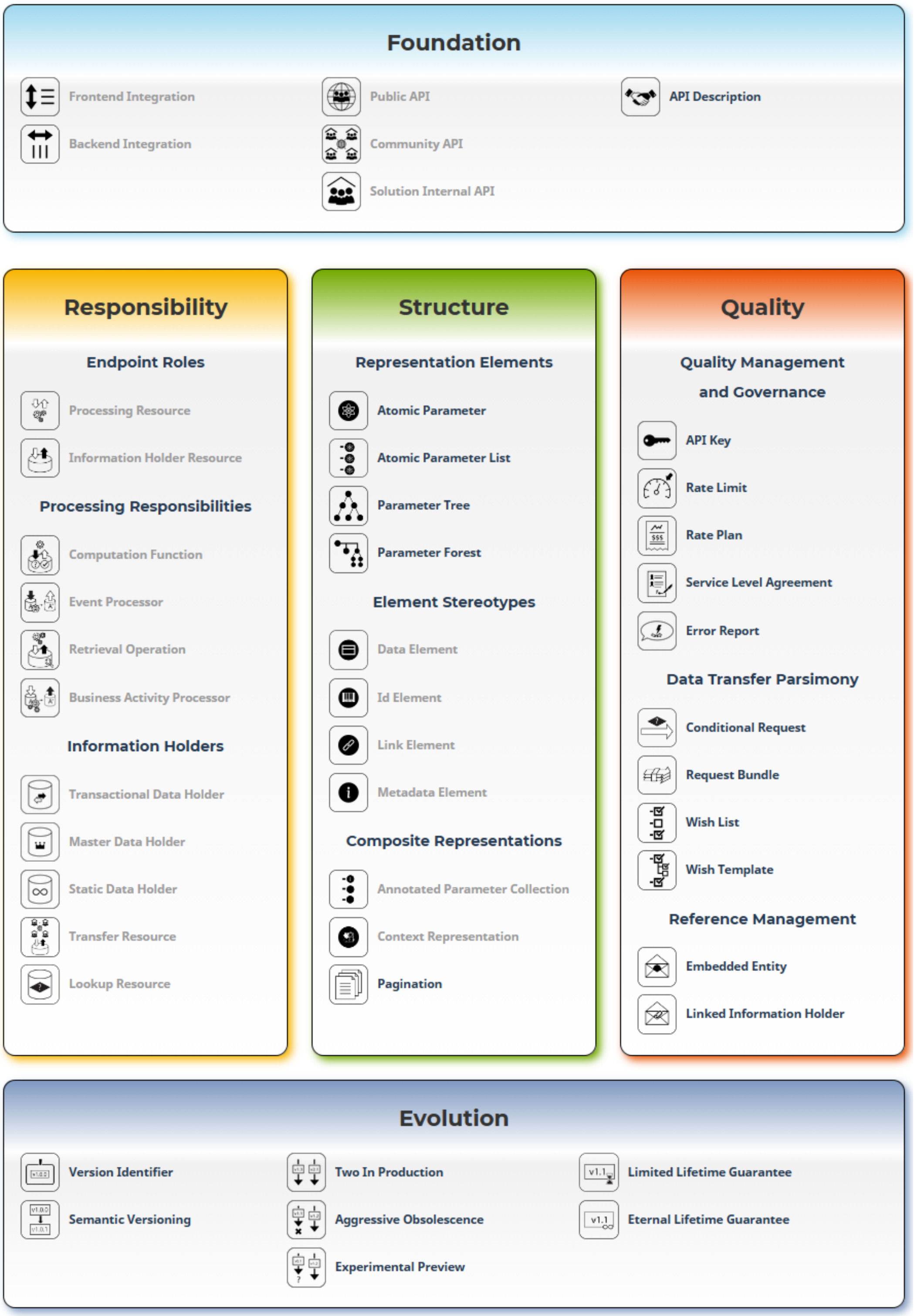


Figure 2 The MAP language is organized into categories, three of which have subcategories. Patterns set in bold/black are already available online at the time of writing; the grayed out ones are currently being mined. The identification category is not available yet. Visit www.microservice-api-patterns.org for an interactive, up-to-date version of this pattern index.

3 Pattern Examples: In-/Excluding Nested Data Representations

In this section, we introduce two patterns from the quality category not featured in peer-reviewed publications yet, *Embedded Entity* and *Linked Information Holder*. They provide two alternatives for representing related data elements: inclusion (nesting) and linkage (referencing).

We use the following template to document all our patterns: The *context* establishes preconditions for pattern eligibility/applicability. The *problem* specifies a design issue to be resolved, typically in question form. The *forces* explain why the problem is hard to solve: architectural design issues and conflicting quality attributes are often referenced here. The *solution* answers the design question introduced by the problem statement, describes how the solution works and which variants (if any) exist. It also gives an example and shares pattern application and implementation hints. The *consequences* section discusses to which extent the solution resolves the pattern forces as well as additional pros and cons; it may also call out new problems or identify alternative solutions. The *known uses* report real-world pattern applications. Finally, the relations to other patterns are explained and additional pointers and references are given under *more information*.

References to other patterns are formatted like this in this paper: *Pattern Name*.

3.1 Pattern: *Embedded Entity*

a.k.a. Inlined Entity Data; Embedded Document (Nesting)

3.1.1 Context

The information required by a communication participant contains structured data. This data includes multiple elements that relate to each other in certain ways. For instance, a master data element such as a customer profile may *contain* other elements providing contact information including addresses and phone numbers, or a periodic business report may *aggregate* source information such as monthly sales figures summarizing individual business transactions. API clients work with several of the related information elements when processing response messages or producing request messages.[2]

3.1.2 Problem

How can you avoid exchanging multiple messages when receivers require insights from multiple related information elements?

3.1.3 Forces

When deciding for or against this pattern, you have to consider its impact on:

- Performance and scalability
- Flexibility and modifiability
- Data quality
- Data freshness and consistency

[2] Note that this is (almost) the same context as in the sibling pattern *Linked Information Holder*.

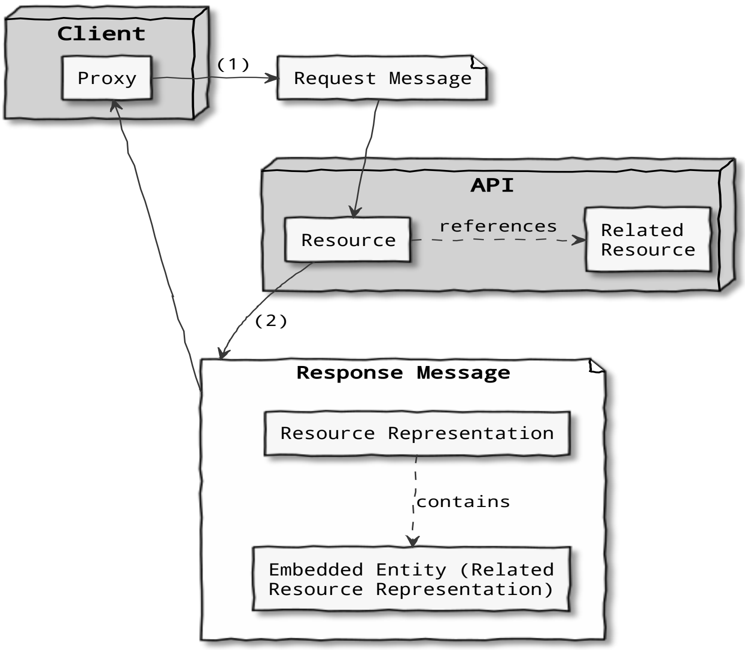


Figure 3 Sketch of Embedded Entity pattern (entities are represented as HTTP resources).

Traversing all relationships between information elements to include all possibly interesting data may require complex message representations and lead to large message sizes. It is unlikely and/or difficult to ensure that all recipients will require the same message content.

3.1.3.1 Non-solution

One could simply define one API endpoint per information element. This endpoint is accessed whenever API clients process data from that information element, e.g., when it is referenced from another one. But if API clients use such data in many situations, this solution leads to many subsequent requests to follow the references. This could possibly make it necessary to coordinate request execution and introduce conversation state, which harms scalability and availability; distributed data also is more difficult to keep consistent than local data.

3.1.4 Solution

For any relationship that the client has to follow, embed an *Entity Element*[3] in the message that contains the data of the target entity (instead of linking to the target entity). For instance, if a purchase order has a relation to product master data, let the purchase order message hold a copy of all relevant information stored in the product master data. Figure 3 shows a solution sketch of *Embedded Entity*.

[3] All patterns that are already published, but not contained in this paper can be found online: `https://microservice-api-patterns.org/`.

3.1.4.1 How it works

Define a *Parameter Tree*[4] or an *Atomic Parameter List* that includes an *Entity Element* for the referenced relationship. Provide an additional *Metadata Element* to denote the relationship type if needed.

Analyze outgoing relationships in the *Entity Element* and consider embedding them in the message as well, but only if this additional data is also used by the API client in enough cases. Repeat this analysis up to reaching the “transitive closure” where all reachable entities have either been included or excluded.

Review each source-target relationship carefully: is the target entity really needed on the API client side in enough cases? A “yes” answer warrants transmitting relationship information as *Embedded Entities*; otherwise transmitting references to *Linked Information Holders* might be sufficient.

Document the existence and the meaning of the embedded entity relationships in the *API Description*.

3.1.4.2 Example

Lakeside Mutual[5], a microservices sample application, contains a service called `Customer Core` that aggregates several information items (here: entities and value objects from Domain-Driven Design) in its operation signatures. An API client can access this data via an HTTP resource API. This API contains several instances of the pattern. Applying the *Embedded Entity* pattern, a response message might look as follows:

```
GET http://localhost:8080/customers/a51a-433f-979b-24e8f0

{
  "customer": {
    "id": "a51a-433f-979b-24e8f0"
  },
  "customerProfile": {
    "firstname": "Robbie",
    "lastname": "Davenhall",
    "birthday": "1961-08-11T23:00:00.000+0000",
    "currentAddress": {
      "streetAddress": "1 Dunning Trail",
      "postalCode": "9511",
      "city": "Banga"
    },
    "email": "rdavenhall0@example.com",
    "phoneNumber": "491 103 8336",
    "moveHistory": [{
      "streetAddress": "15 Briar Crest Center",
      "postalCode": "",
      "city": "Aeteke"
    }]
  },
```

[4] See `https://microservice-api-patterns.org/`.

[5] `https://github.com/Microservice-API-Patterns/LakesideMutual`

```
  "customerInteractionLog": {
    "contactHistory": [],
    "classification": "??"
  }
}
```

The referenced information items are all fully contained in the response message (e.g., `customerProfile`, `customerInteractionLog`); no URIs (links) to other resources appear. Note that `customerProfile` actually embeds nested data (`currentAddress`, `moveHistory`), while the `customerInteractionLog` is empty in this exemplary data set.

3.1.4.3 Implementation hints

When embedding entity relationships in message representations, keep in mind to:

- Document data characteristics such as owner, provenance, lifetime, and last update in the *API Description*; consider to introduce corresponding *Metadata Elements* if the data is used by a sufficient amount of clients requiring additional explanations.
- Distinguish transactional data from master data and other reference data when embedding it (to account for their different life cycles, evolution roadmaps and validity timeframes).
- Secure the message so that the content part with the highest protection need is covered adequately; this might (or might not) be the *Embedded Entity.* If the security requirements of link source and target differ substantially, consider switching to the sibling pattern *Linked Information Holder.*
- Be careful with consumer-side caching and replicating parts or all of the embedded data as this may introduce consistency, concurrency, and/or data ownership issues, especially when mixing transactional data with master data in one message.
- Test the use of *Embedded Entities* with all valid and invalid cardinalities. More specifically, empty, one, few, or many referenced data items should appear in different test cases.
- Monitor message sizes at runtime to prepare for interface refactoring such as switching to *Linked Information Holder* (see discussion below) or introducing *Pagination.*
- Define compatibility rules and service evolution policies [11] when introducing *Embedded Entites.* The more related entities a message includes and the more complex its payload is, the more likely it is to change as a whole and in parts. As a client, behave as a *Tolerant Reader* [2]: Clients should not assume that all related entities will always be included and might have to be ready to follow a link in case the information is not embedded.

3.1.5 Consequences

The pattern meets the “all in one” requirement articulated by the problem statement, but this may lead to large messages that are expensive to transfer. If some clients do not have to receive all the data, then parts of the payload could have been omitted.

3.1.5.1 Resolution of forces

+ An *Embedded Entity* reduces the number of calls required: If the required information is included, the client does not have to create a request to obtain it.
+ Embedding entities can lead to a reduction in the number of endpoints, because no dedicated endpoint to retrieve some information is required.

– Embedding entities leads to larger response messages which take longer to transfer.

- It can be difficult to anticipate what information different clients require to perform their tasks. As a result, there is a tendency to include more data than needed by (most) clients in an *Embedded Entity*, which leads to yet larger message sizes. Such design can be found in many *Public APIs* serving large and possibly unknown clients.
- Large messages that contain unused data consume more bandwidth than necessary. However, if most or all of the data is actually used, sending many small messages might actually require more bandwidth than sending one large message (e.g., for header and metadata sent with the smaller messages multiple times).
- If the embedded entities change with different speed (e.g., a fast-changing transactional entity refers to immutable master data), retransmitting all entities causes unnecessary overhead as messages with partially changed content cannot be cached. Consider switching to a *Linked Information Holder* (and maybe additionally apply the *Conditional Request* pattern for the linked entity).
- Once included and exposed in an *API Description*, it is hard to remove an *Embedded Entity* in a backward-compatible manner (as clients may have begun to rely on it).

3.1.5.2 Alternatives

If reducing message size is your main design goal, you can also define a *Wish List* or, even more expressive, a *Wish Template* to minimize the data to be transferred by letting consumers dynamically describe which subset of the data they need.

API Gateways[6] can also help when dealing with different information needs. They can either provide two alternative APIs that use the same backend interface, and/or collect and aggregate information from different endpoints and operations (which makes them stateful).

3.1.6 Known Uses

Many public APIs with complex response messages use the *Embedded Entity* pattern:

- When retrieving an issue with the GitHub v3 API[7], the response also contains the full information about the milestone the issue is assigned to.
- A tweet in the Twitter REST API[8] contains the entire user information, including for example the number of followers the user has.
- Many operations in the Microsoft Graph API apply this pattern. For instance, the user resource representations[9] contain structured attributes that represent (sub-)entities (but also link to other resources via *Linked Information Holders*). For instance, the response body of `List events` contains an array of `attendees` that are identified by their email addresses, but also have a `type` and a `status`.

Plenty of APIs offered by custom enterprise information systems and master data management products also realize the pattern.

[6] https://microservices.io/patterns/apigateway.html

[7] https://developer.github.com/v3/issues/#get-a-single-issue

[8] https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/get-statuses-show-id

[9] https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0/resources/user

3.1.7 More Information

3.1.7.1 Related Patterns

Linked Information Holder describes a complementary solution for the reference management problem and can also be seen as an alternative (as explained above).

Wish List or *Wish Template* can help to fine-tune the content in an *Embedded Entity*, as explained above.

3.1.7.2 Other Sources

See Section 7.5 in [18] for additional advice and examples (“Embedded Document (Nesting)”).

3.2 Pattern: *Linked Information Holder*

a.k.a. Linked Entity, Data Reference; Compound Document (Sideloading)

3.2.1 Context

An API exposes structured data to meet the information needs of the communication participants. This data contains elements that relate to each other (e.g., a master information element may *contain* other elements providing detailed information or a performance report for a period of time may *aggregate* raw data such as individual measurements). API clients want to work with several of the related information elements when processing response messages or producing request messages.[10]

3.2.2 Problem

When exposing structured, possibly deeply nested information elements in an API, how can you avoid sending large messages containing lots of data that is not always useful for the message receiver in its entirety?

3.2.3 Forces

A general rule of thumb in distributed systems is that request and response messages should not be too large so that the network and the endpoint processing resources are not over-utilized. That said, message recipients possibly would like to follow many or all of the relationships to access information elements related to the elements requested. If the related elements are not included, information about their location and their content is required, as well as access information. This information set has to be designed, implemented and evolved; the resulting dependency has to be managed.

The sibling pattern *Embedded Entity* list additional forces that apply to both patterns.

[10] This context is (almost) the same as that of the sibling pattern *Embedded Entity*.

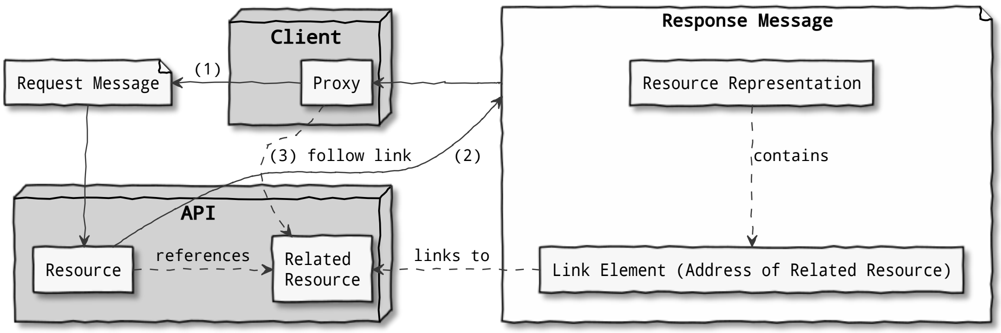


Figure 4 Sketch of Linked Information Holder pattern.

3.2.3.1 Non-solution

One option is to always (transitively) include all the related information elements of each transmitted element in request and response messages throughout the API (as described in the *Embedded Entity* pattern). However, this approach can harm performance of individual calls and lead to large, wasteful messages containing data not required by some clients.

3.2.4 Solution

Add a *Link Element*[11] to the message that references an API endpoint. Introduce an API endpoint that represents the linked entity, for instance, an *Information Holder Resource* for the referenced information element. This element might be an entity from the domain model[12] that is exposed by the API. Figure 4 outlines this solution.

3.2.4.1 How it works

Include the location information (i.e., host and port), expressed in the logical naming and/or addressing scheme of the API, when referencing the endpoint via *Link Elements* in request and response messages. This typically requires a *Parameter Tree* to be used in the representation structure; in simple cases, an *Atomic Parameter List* might suffice.

Identify the *Link Element* with a name. If more information about the relation should be sent to clients, annotate this *Link Element* with details about the corresponding relationship, for instance, a *Metadata Element* specifying the type and characteristics of the relationship. Clients and providers must agree on the semantics (i.e., meaning) of the link relationships, and be aware of coupling and side effects introduced.

Document the existence and the meaning of the *Linked Information Holder* in the *API Description*. Specify the cardinalities on both ends of the relation. One-to-many relationships can be modeled as collections, for instance by transmitting multiple elements as *Atomic Parameter Lists*. Many-to-many relationships can be modeled as two such one-to-many relationships, one linking the source entities to the targets, and one linking the target entities to the sources. Such design may require the introduction of an additional API endpoint representing the relation rather than its source and target.

[11] See `https://microservice-api-patterns.org/`.
[12] `https://en.wikipedia.org/wiki/Domain_model`

3.2.4.2 Example

A sample application for `Customer Management` could work with a `Customer Core` service API that aggregates several information elements from the domain model of the application, in the form of entities and value objects from Domain-Driven Design (DDD). Its API client could access this data through a `Customer Information Holder`, implemented as a REST controller in Spring Boot.

If the `Customer Information Holder` implements the *Linked Information Holder* pattern for the `customerProfile`, a response message looks as this:

```
GET http://localhost:8080/customers/a51a-433f-979b-24e8f0

{
  "customer": {
    "id": "a51a-433f-979b-24e8f0"
  },
  "links": [{
    "rel": "customerProfile",
    "href": "http://localhost:8080/customers/a51a-433f-979b-24e8f0/profile"
  }, {
    "rel": "moveHistory",
    "href": "http://localhost:8080/customers/a51a-433f-979b-24e8f0/moveHistory"
  }],
  "email": "rdavenhall0@example.com",
  "phoneNumber": "491 103 8336",
  "customerInteractionLog": {
    "contactHistory": [],
    "classification": "??"
  }
}
```

The `customerProfile` can then be retrieved by a subsequent GET request to the provided URI link. The `moveHistory` has been factored out as well, so the pattern is applied twice in this example.

3.2.4.3 Implementation hints

When adding links to message representations to turn relationship targets into API endpoints, it is good practice to:

- Keep the naming scheme and structure of the *Link Elements* consistent and be reluctant to change it. For instance, keep the URI naming scheme consistent in HTTP resource APIs.
- Define *compliance controls* if the link relations are subject to system and process assurance audits as discussed in [8].[13]
- Run regression tests on the source of a link when the link destination changes its interface or implementation.

[13] An example of such a control is a pre- and postcondition check at the API endpoint boundary that enforces the correct cardinality of a link from a purchase order to customer (e.g., there has to be one and only one customer per order). Such design-by-contract approaches can be implemented as a foreign key relationship if both order and customer are stored in the same relational database. In a distributed services architecture, both sides of the relationship can modify the data independently of each other, which might introduce inconsistencies.

- Monitor performance continuously to preserve and challenge the rationale for pattern usage. If most or all client calls follow the given link, consider embedding the target element in the original representation to reduce traffic (see *Embedded Entity* pattern).
- Adhere to REST constraints and related recommended practices when using RESTful HTTP (see [1]): Linked reference data is a cornerstone of the Hypertext as the Engine of Application State (HATEOAS) tenet that is required to reach REST maturity level 3[14].

3.2.5 Consequences

This pattern is often applied when referencing rich information holders serving multiple usage scenarios: not all message recipients require the full set of referenced data, for instance, when *Master Data Holders* such as customer profiles or a product records are referenced. Following the link, message recipients can obtain the required subsets on demand.

When introducing link elements into message representations, an implicit promise is made to the recipient that these links can be followed successfully; the provider might not be willing to keep such promise infinitely. Even if a long lifetime of the linked endpoint is guaranteed, links still may break; for instance, when the data organization or location changes. Clients should expect this and be able to follow redirects or referrals to the updated links. To minimize breaking links on the provider side, the provider should invest in maintaining link consistency. For instance, a *Lookup Resource* can be used to solve this problem.

3.2.5.1 Resolution of forces

+ Linking instead of embedding entities results in smaller messages and uses less resources in the communications infrastructure when exchanging individual messages. This needs to be contrasted to the possible higher resource use due to transfer of multiple messages as links get followed.

+ If some linked data changes often, only that data needs to be requested again.

– Additional requests are required to dereference the linked information.

– Linking instead of embedding entities might result in the use of more resources in the communications infrastructure as multiple messages are required to follow the links.

– Additional *Information Holder Resource* endpoints have to be provided for the linked entities, causing development and operations effort and cost.

3.2.5.2 Alternatives

The patterns in the *Quality Patterns* category that help reduce the amount of data exchanged can be used alternatively. For instance, *Conditional Request*, *Wish List*, and *Wish Template* are eligible; the structure pattern *Pagination* is an option too.

3.2.6 Known Uses

Many public Web APIs apply this pattern, for instance the JIRA Cloud REST API[15] when reporting the links between issues in the Get issue link[16] call.

[14] https://martinfowler.com/articles/richardsonMaturityModel.html
[15] https://docs.atlassian.com/jira/REST/cloud/
[16] https://developer.atlassian.com/cloud/jira/platform/rest/#api-api-2-issueLink-linkId-get

Certain calls in the Microsoft Graph API also apply this pattern: for instance, user resource representations[17] contain scalar and complex attributes as "Properties", but also link to other resources such as Calendar (under "Relationships").

RESTful HTTP APIs on maturity level 3 apply this pattern if the links representing application state transfer deal with both master data and transactional data resources and their representations. An example is Spring Restbucks[18].

3.2.7 More Information

3.2.7.1 Related Patterns

The sibling pattern *Embedded Entity* provides an alternative to *Linked Information Holder*, transmitting related data rather than referencing it.

Linked Information Holders typically reference *Information Holder Resources*. The referenced *Information Holder Resources* can be combined with *Lookup Resource* to cope with potentially broken links, as outlined previously.

Linked Service[19] is a similar pattern in [2], but less focused on data. "Web Service Patterns" [12] has a *Partial DTO Population* pattern which solves a similar problem.

3.2.7.2 Other Sources

See Section 7.4 in [18] for additional advice and examples, to be found under "Compound Document (Sideloading)".

The Backup, Availability, Consistency (BAC) theorem investigates data management issues further [14].

4 Conclusion

This paper introduced Microservice API Patterns (MAP), a volunteer project compiling a pattern language for the design and evolution of Microservice APIs. The language is organized into six pattern categories at present; the MAP website[20] provides additional navigation aids such as a cheat sheet and pattern filtering by scope, phase, role, and quality attributes. The patterns, their known uses and the examples have been mined from public Web APIs as well as application development and software integration projects the authors and their industry partners have been involved in [21]. We previously published 18 patterns at pattern conferences; this paper introduced two more.

In our future work, we plan to fill gaps throughout our six pattern categories. The next patterns will describe additional structural representations as well as the architectural roles and responsibilities of endpoints and operations within an API. Patterns capturing API endpoint and service identification strategies and tactics as well as corresponding artifacts yet have to be mined: Context Maps, Bounded Contexts and Aggregates from Domain-Driven Design (DDD) [3] seem to be particularly promising starting points for microservice API design, and tools for strategic DDD are beginning to emerge [9]. We have also begun to work on a technology-independent service contract language that incorporates our patterns as first-class language elements, as well as tools to create API specifications from DDD context maps, existing code, and as other specification languages such as Open API Specification. Other future tools may search for pattern instances and provide metrics.

[17] https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0/resources/user
[18] https://github.com/odrotbohm/spring-restbucks
[19] http://www.servicedesignpatterns.com/ClientServiceInteractions/LinkedService
[20] https://microservice-api-patterns.org/

References

1 Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

2 Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, 2011. URL: `http://www.servicedesignpatterns.com/`.

3 Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, 2003.

4 Pat Helland. Data on the Outside Versus Data on the Inside. In *CIDR 2005, Second Biennial Conf. on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 144–153, 2005. URL: `http://cidrdb.org/cidr2005/papers/P12.pdf`.

5 Gregor Hohpe. SOA patterns: New insights or recycled knowledge? `https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf`, 2007. URL: `https://www.enterpriseintegrationpatterns.com/docs/HohpeSOAPatterns.pdf`.

6 Gregor Hohpe, Rebecca Wirfs-Brock, Joseph W. Yoder, and Olaf Zimmermann. Twenty Years of Patterns' Impact. *IEEE Software*, 30(6):88, 2013. `doi:10.1109/MS.2013.135`.

7 Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

8 Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. Compliance by design–Bridging the chasm between auditors and IT architects. *Computers & Security*, 30(6):410–426, 2011.

9 Stefan Kapferer. A Domain-specific Language for Service Decomposition. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018. URL: `https://eprints.hsr.ch/722`.

10 James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html/`, 2014. URL: `https://martinfowler.com/articles/microservices.html/`.

11 Daniel Lübke, Olaf Zimmermann, Mirko Stocker, Cesare Pautasso, and Uwe Zdun. Interface Evolution Patterns - Balancing Compatibility and Extensibility across Service Life Cycles. In *Proc. of the 24th European Conference on Pattern Languages of Programs*, EuroPLoP '19, 2019.

12 Paul B. Monday. *Web Services Patterns: Java Edition*. Apress, Berkely, CA, USA, 2003.

13 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

14 Guy Pardon, Cesare Pautasso, and Olaf Zimmermann. Consistent Disaster Recovery for Microservices: the BAC Theorem. *IEEE Cloud Computing*, 5(1):49–59, December 2018. `doi:10.1109/MCC.2018.011791714`.

15 Cesare Pautasso and Olaf Zimmermann. The Web as a Software Connector: Integration Resting on Linked Resources. *IEEE Software*, 35:93–98, 2018. `doi:10.1109/MS.2017.4541049`.

16 Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, 34(1):91–98, 2017. `doi:10.1109/MS.2017.24`.

17 Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Interface Quality Patterns - Communicating and Improving the Quality of Microservices APIs. In *Proc. of the 23nd European Conference on Pattern Languages of Programs*, EuroPLoP '18, 2018.

18 Phil Sturgeon. *Build APIs you won't hate*. LeanPub, https://leanpub.com/build-apis-you-wont-hate, 2016.

19 Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

20 Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education, 2002.

21 Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *16th International Conference on Service-Oriented Computing ICSOC 2018*, pages 78–89, November 2018. URL: `http://eprints.cs.univie.ac.at/5956/`.

22 Olaf Zimmermann. Microservices Tenets. *Comput. Sci.*, 32(3-4):301–310, July 2017. `doi:10.1007/s00450-016-0337-0`.

23 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proc. of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, pages 27:1–27:36. ACM, 2017. `doi:10.1145/3147704.3147734`.

PREvant (Preview Servant): Composing Microservices into Reviewable and Testable Applications

Marc Schreiber
aixigo AG, Aachen, Germany
marc.schreiber@fh-aachen.de

Abstract

This paper introduces PREvant (preview servant), a software tool which provides a simple RESTful API for deploying and composing containerized microservices as reviewable applications. PREvant's API serves as a connector between continuous delivery pipelines of microservices and the infrastructure that hosts the applications. Based on the REST API and a web interface developers and domain experts at aixigo AG developed quality assurance workflows that help to increase and maintain high microservice quality.

2012 ACM Subject Classification Software and its engineering → Software creation and management; Software and its engineering

Keywords and phrases Microservice development, testing for microservices, exploratory testing, development workflows

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.5

Supplement Material

- The source code is available on GitHub: `https://github.com/aixigo/PREvant`
- A short talk about PREvant is available on YouTube: `https://www.youtube.com/watch?v=O9GxapQR5bk`

1 Introduction

Currently, an increasing number of enterprises develop microservices to build their applications [2, 11, 18] because microservices are scalable and offer quick deployment cycles, superior quality, and greater flexibility compared to monolithic software [21, 24]. Furthermore, numerous companies migrate their on-premises applications to microservice architectures [13] because of the aforementioned advantages combined with agile software development. However, when development teams begin to build microservices, they can face major challenges due to the increased cognitive load, design complexity, testing and maintenance efforts [30].

Microservices must be deployed on an infrastructure to perform automatic and user-based acceptance tests, thereby ensuring feature correctness. Developers can employ containerization techniques to package and deploy their microservices [17]; however, they must manage the complexity of deployment set-ups when they provide the whole application as a preview, which consists of multiple microservices distributed across numerous source-code repositories. To deploy their services to a container orchestration platform for testing purposes, developers must create complex continuous delivery pipelines that utilize container orchestration platforms. In addition to these technical challenges, the testing of microservices provides further challenges that require effective testing strategies [15, 22].

This paper introduces *PREvant (Preview Servant)*, a software tool that helps to deliver high-quality microservices by providing an approach to deploying and composing containerized microservices as reviewable applications. PREvant simplifies the deployment of applications that comprise multiple microservices, and it supports various configuration scenarios that

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 5; pp. 5:1–5:16
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

equip the application with the necessary infrastructure through companions. Thanks to these and additional features, developers and domain experts can employ established workflows to ensure that their teams are building high-quality services. Additionally, sales and team managers can utilize PREvant's capabilities.

The remainder of this paper is structured as follows: Section 2 provides basic concepts and technologies that are necessary for understanding PREvant's use cases. Additionally, this section presents related work. Section 3 provides an exemplary case study to support PREvant's use cases with a meaningful example, and Section 4 depicts PREvant's approach, implementation, and architecture. Section 5 illustrates established workflows utilized at aixigo AG that ensure high quality microservices. Section 6 supplies recipes that have been collected through the utilization of PREvant at aixigo AG. Finally, Section 7 concludes the paper.

2 Background and Related Work

The development of microservices is based on the following characteristics [18]:

> Microservices are small, autonomous services that work together ... and [a service] might be deployed as an isolated service on a platform as a service (PAAS), or it might be its own operating system process. ... All communication between the services themselves are via network calls, to enforce separation between the services and avoid the perils of tight coupling. ... [The] service exposes an application programming interface (API), and collaborating services communicate with us via those APIs.

Therefore, developers who create applications consisting of independent microservices must ensure that the services provide feature correctness even when the independent microservices do not share any resources such as operating systems or databases. Due to the independence of microservices, Docker [17] and standard containerization techniques [9] support the development of such microservice architecture because Docker and containers in general suit each other in implementing this kind of architecture [14]. This approach of implementing microservice architectures [9, 14, 17] provides the foundation of PREvant's approach.

Because microservice architectures rely on unreliable network calls that require fault-tolerant services [18, Chapter 11], developers must ensure that the source code includes this fault tolerance. Therefore, developers can rely on automated unit tests and continuous integration to ensure high-quality code [12]. Additionally, developers must ensure that the whole application works as expected, a factor which is often confronted by continuous delivery pipelines [4]. Figure 1 depicts a continuous delivery pipeline for a microservice as described by Chen [4].

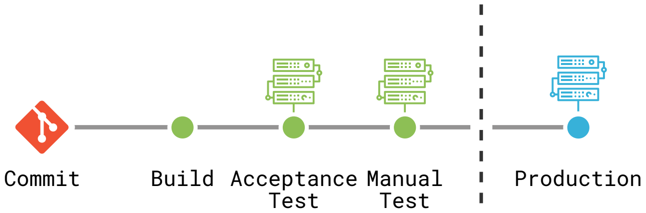


Figure 1 Continuous Delivery Pipeline Stages.

The build stage in Figure 1 represents the continuous integration stage, in which the microservice is compiled, tested, and packaged into a runnable format, such as a Docker container image [9, 17]. When the build stage is completed, the continuous delivery pipeline executes the next stages:

Acceptance Test In this stage, automated end-to-end tests ensure that the microservice performs as expected. For example, an automated browser test ensures that the microservice's data is rendered correctly.

Manual Test This stage provides a running instance of the microservice for domain experts who perform exploratory testing to ensure correct business behavior.

Production When the domain experts establish that the microservice performs correctly, the service can be deployed into production, completing the continuous delivery pipeline.

The acceptance and manual test stages include environments similar to the production environment in which the microservices will be deployed (depicted by the infrastructure symbols above the stages in Figure 1). The deployment in these environments is handled by the continuous delivery pipeline and is supported by different development tools, such as GitLab Review Apps[1] or GitOps,[2] that utilize container orchestrations to spin the required environments. Such an approach [4] builds common ground for GitLab Review Apps, GitOps, and PREvant.

However, existing solutions have a common disadvantage when developers build multiple microservices in multiple source-code repositories, a factor which is addressed by PREvant. Existing solutions such as GitLab Review Apps do not manage effectively when the development of an application consists of multiple microservices distributed across multiple source-code repositories, as depicted in Figure 2. In this scenario, the continuous delivery pipelines of each microservice repository (see the commit stage) must be aware of foreign microservices and ensure that their environments include them (see question marks in Figure 2). Therefore, both GitLab Review Apps and GitOps offer to write deployment scripts that ensure that the foreign microservices are deployed as well, which generates a tight coupling between the deployment pipelines, thus violating the mantra of independent microservice architecture.

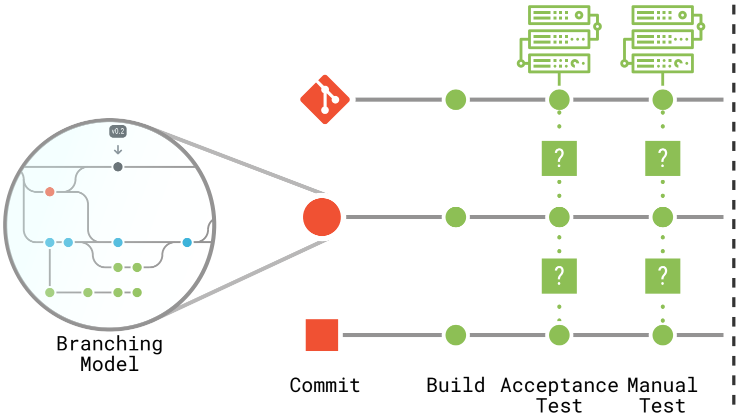


Figure 2 Continuous Delivery Pipeline Stages with Multi-Repository Development.

[1] https://about.gitlab.com/product/review-apps/
[2] https://www.weave.works/technologies/gitops/

Furthermore, the development of each microservice could follow a branching model, as depicted on the left-hand side of Figure 2, and each branch should be tested through the same stages of the delivery pipeline as the mainline branch. However, if a feature requires the extensions of two or more microservices, then this must be configured and written into the source code of the deployment scripts, which requires thorough clean-up afterward. This clean-up could fail and break the deployment of the mainline branches.

In contrast to existing solutions, PREvant aims to improve the handling of delivery pipelines in multi-repository, multi-branch scenarios, some of which are described in further detail in Section 4. Therefore, PREvant relies on following techniques to compose a set of microservices into one application:

1. Because PREvant supports the development of microservices, and containers are an ideal match for microservice architecture [14], PREvant relies on container runtime infrastructure, such as Docker, Docker Swarm, or Kubernetes to deploy the microservices into staging areas.[3]
2. To compose the microservices on a container runtime, PREvant relies on a container registry to exchange the container images between a continuous delivery pipeline and the container runtime.
3. Additionally, the container runtimes provide software-defined networking [7] that PREvant utilizes to isolate the microservice applications from each other.
4. To make the composed applications accessible to the domain experts, PREvant equips the containers in such a way that Traefik [5] can work as a reverse proxy, which makes Traefik a requirement for PREvant.

3 Case Study

To support PREvant's use cases with an illustrative example, this section introduces a sample e-commerce system that offers end customers the ability to purchase products in a web shop. The example is borrowed from Wolff [29] and provides the following services:

order This microservice provides a web interface that accepts orders through a shopping cart. All accepted orders are stored in a database and are published through an asynchronous messaging channel.

invoice This service subscribes to the orders channel and extracts all relevant information from the messages. The relevant information is stored as new invoices in a database, and the accounting department receives new invoices through the invoice service's web interface.

shipping Similar to the invoice service, the shipping service subscribes to the orders channel and stores the relevant shipping information in a database. The shipping department receives new shipping requests through the shipping service's web interface.

In the context of this paper, the development of these services is distributed across three source-code repositories, and each repository manages a continuous delivery pipeline that ensures that the services perform as desired. Additionally, the development team is supported through domain experts who randomly perform exploratory testing on the whole e-commerce system to ensure that the system works well for the salespeople who operate the system.

[3] Currently, PREvant only supports Docker, but PREvant's developers plan to support Kubernetes in the future.

Therefore, the continuous delivery pipelines must ensure that domain experts can access the application at any time, as depicted in Figure 2.

Furthermore, the microservices rely on infrastructure services. They require a Apache Kafka[4] instance to exchange the order message and PostgreSQL[5] as a database instance to store the service-relevant information. These infrastructure services must be deployed through the continuous delivery pipelines as well to ensure a running application.

4 Composing Microservices with PREvant

As stated in Section 2, PREvant aims to simplify the composition of microservices through continuous delivery pipelines. Therefore, PREvant serves as a connector between the continuous delivery pipelines and the infrastructure that hosts the applications for testing and quality-assurance purposes, as depicted in Figure 3. This approach facilitates the composition of reviewable applications that are explained in detail in this section.

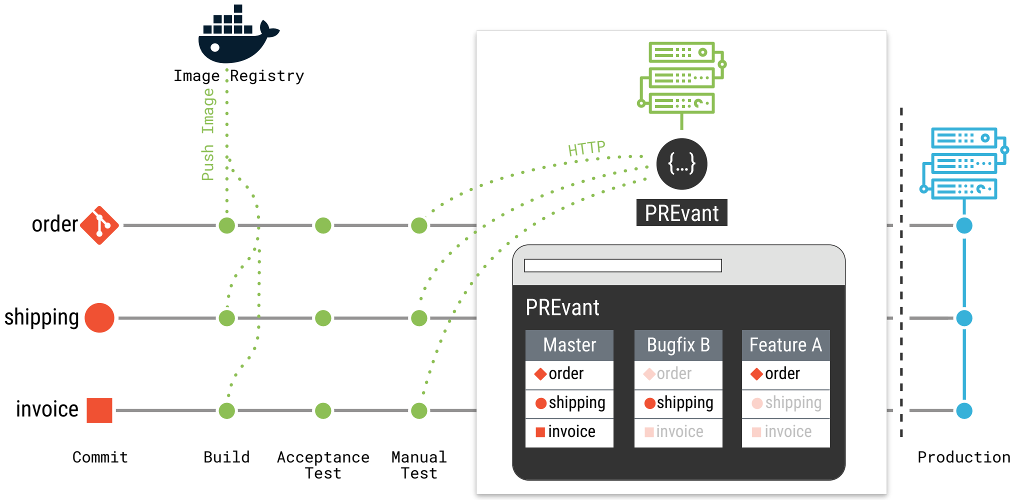


Figure 3 Composing Microservices with PREvant into Reviewable Applications.

Figure 3 illustrates the disjoint repositories and continuous delivery pipelines of the microservices *order*, *shipping*, and *invoice*. The build stage packages the microservices as a container image and pushes it to a container image registry (e.g. a Docker registry) to ensure that the services are ready for deployment in the acceptance and manual test stages. In a deployment phase, such as the manual test stage depicted in Figure 3, the continuous delivery pipeline can utilize PREvant's REST API, as illustrated in Listing 1. This REST request creates a software-defined network [7], initiates the container for the microservice, connects it to the network, and creates a reverse-proxy configuration, making the service accessible through PREvant's web interface.[6] Subsequent REST calls check whether the container image has a newer version, and if so, then the container is updated.

[4] `https://kafka.apache.org/`
[5] `https://www.postgresql.org/`
[6] In this example the invoice service is available through the relative URL `/master/invoice`.

Listing 1 Deploy "invoice" Service.

```
POST /api/apps/master HTTP/1.1
Content-Type: application/json
Accept: application/json

[{
  "serviceName": "invoice",
  "image": "registry.example.com/a-team/invoice:master",
}]
```

When the continuous delivery pipelines of the remaining microservices utilize a similar REST call, for example, replacing `invoice` in Listing 1 with `order` or `shipping` PREvant initiates the containers and connects them to the existing software-defined network so that the services can communicate with each other. Then, domain experts can access the services through PREvant's web interface, and they can begin exploratory testing before the microservices are deployed to production. Through this approach, PREvant offers the following key concepts:

- PREvant's REST API does not expose the internal workings of the underlying container runtime, which makes the deployment infrastructure agnostic. By design, the software architecture of PREvant employs an infrastructure abstraction so that Docker and Kubernetes are supported platforms; however, PREvant is not limited to these container orchestration platforms. Section 4.1 provides an architectural overview.
- The REST API approach reduces the complexity of continuous delivery pipelines because the necessity of deployment scripts for specific container orchestration is eliminated.
- The REST API approach enables further use cases that are employed in development workflows, as illustrated in Section 5.
- PREvant supports different microservice architectures so that it is not limited to a specific kind of microservice architecture. This is implemented through the concept of companions, as explained in Section 6.

In addition to these key concepts, PREvant supports feature branch workflows across multiple build pipelines and is not limited to mainline branches of microservices. As illustrated in Figure 2 in Section 2, a feature-based branching model raises the following issue: how does one deploy or compose the whole application automatically when developing a new feature on a branch for a single microservice? For this use case, PREvant provides following solution:

While PREvant can compose the mainline branches of multiple microservices into one application, it also utilizes the mainline branch as a template for each feature branch, as illustrated in Figure 4. To provide a fully functional application that can be reviewed by the domain experts, PREvant replicates all missing services from the master application. For example, if the delivery pipeline executes a POST request to deploy the service `order`, then PREvant compares the set of running microservices in the master application with the provided set of microservices. Then it includes the microservices that are not included and initiates them to provide a fully functional application. Here, it would deploy new container instances of the container images for the services `shipping` and `invoice`.

To distinguish the applications, the deployment pipeline must choose the names that are defined by a path parameter at the REST API level, as illustrated in Listing 2. Here, the application named `feature-a` (see path parameter) is deployed with the container image of the `order` service that has been labeled with `feature-a`. These names can be derived from the branch names through the continuous delivery pipeline, which is an established pattern at aixigo AG.

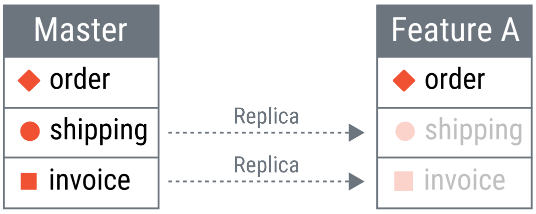


Figure 4 Replication of Services for Feature Branch Workflows.

Listing 2 Deploy Feature Branch of `"order"` Service.

```
POST /api/apps/feature-a HTTP/1.1
Content-Type: application/json
Accept: application/json

[{
  "serviceName": "order",
  "image": "registry.example.com/a-team/order:feature-a",
}]
```

Additionally, if a feature requires changes in two services, then the feature branches can use the same application names to deploy and test the feature across multiple microservice.

4.1 Architecture

To provide the aforementioned use cases, PREvant is implemented as a self-contained system [28] written in Rust [16]; it works in conjunction with Traefik and a container runtime, as illustrated in Figure 5. PREvant's architecture is divided into the following layers.

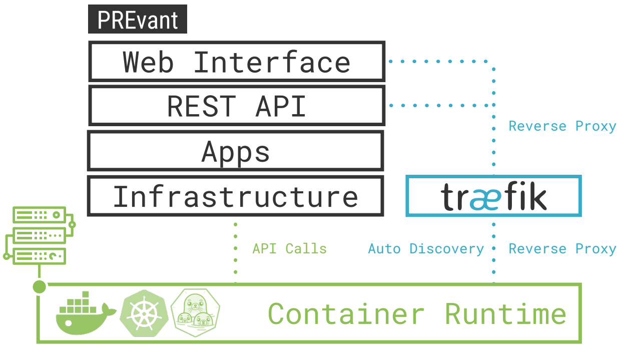


Figure 5 PREvant's Architecture.

Web Interface To provide the use case so that domain experts can access the reviewable applications, PREvant offers a web interface that employs PREvant's REST API to render the available applications. This interface is implemented as a single-page application written in Vue.js,[7] and Figure 6 displays the result in a screenshot of the web interface.

[7] `https://vuejs.org/`

REST API As an interface for continuous delivery pipelines and the web interface, PREvant implements its REST API with Rocket,[8] which is a web framework for Rust. This API layer forwards HTTP requests to the Apps layer.

Apps This layer implements the logic of PREvant's use cases. For example, the request to deploy a microservice is enriched with further information, such as additional services that must be deployed (see Figure 4), and the enriched information is passed to the Infrastructure layer.

Infrastructure The Infrastructure layer serves as a connector between the actual container runtime and the Apps layer by translating requests from the Apps layer into API calls to the container runtime. For example, when PREvant utilizes Docker as a container runtime, the request to list all running applications with the running services is translated into the corresponding API call,[9] as illustrated through *API Calls* in Figure 5. Further requests from the Apps layer are translated as well.

Additionally, this layer is responsible to create the software-defined network for every application so that the microservices of one application can communicate with each other. Additionally, this layer assigns DNS names to the services that are equivalent to the value of the field `"serviceName"` in the REST request.

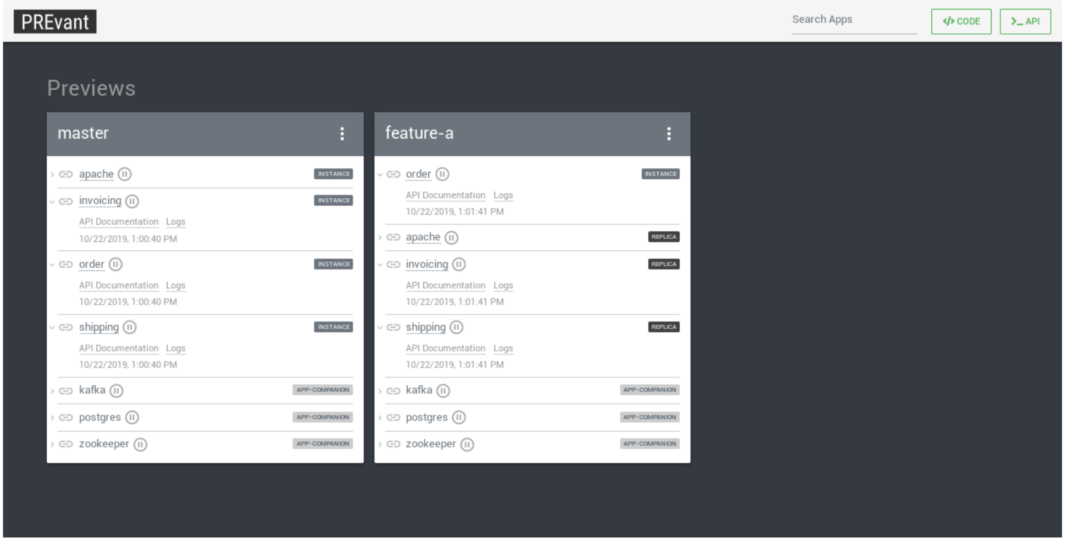


Figure 6 PREvant's Web Interface.

While this architecture solves the deployment and composition problems, it does not serve a proxy mechanism to make the microservices accessible. Therefore, PREvant utilizes Traefik's capabilities [5] as a reverse proxy. PREvant configures the running containers through the Infrastructure layer in such a way that Traefik can utilize the automatic discovery of containers to provide HTTP routes to the microservices, as illustrated in Figure 7. The HTTP requests of each web client, such as a browser, are routed through Traefik, which determines the microservice (concurrently running on the infrastructure) responsible to process the request. Additionally, Traefik routes all requests to PREvant's API and web interface.

[8] `https://rocket.rs/`

[9] `https://docs.docker.com/engine/api/v1.40/`

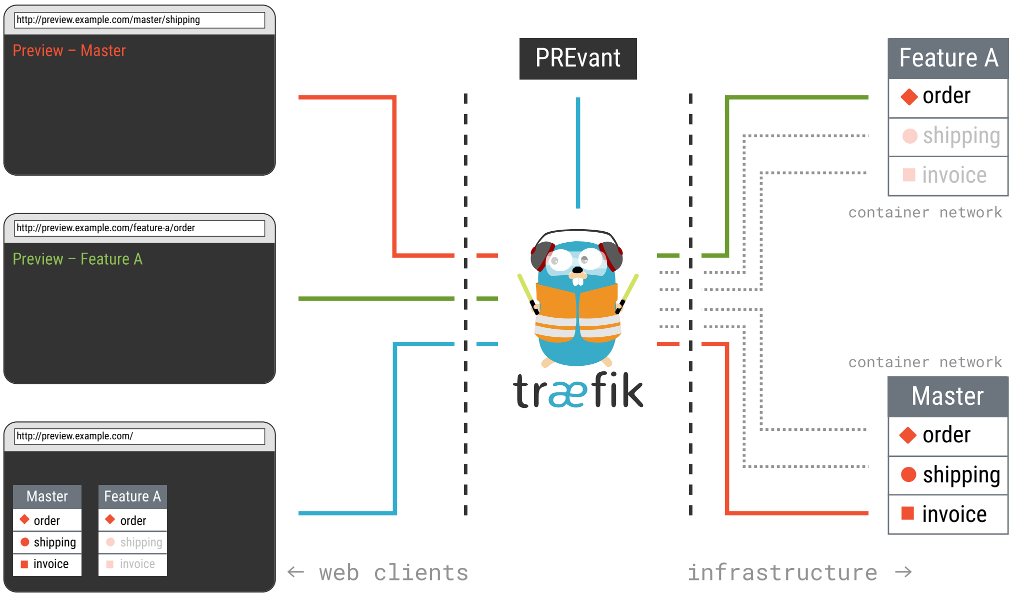


Figure 7 Traefik Serving the Microservices.

To utilize Traefik's capabilities, PREvant must label each microservice with routing information to enable Traefik to discover the microservices automatically. For example, if PREvant is hosted on `http://preview.example.com/`, then the `invoice` service of the application `master` would be configured to be accessible at following URL, as shown in Figure 7: `http://preview.example.com/master/invoice`

4.2 Conventions on Microservices and Application

As illustrated above, PREvant provides a web interface that makes PREvant's applications accessible to the domain experts, sales and team managers, and developers, as illustrated in Figure 6 which displays the microservices of the fictitious e-commerce shop in Section 3. This interface provides following features:

- Access to the microservices via HTML links, which are accessible through Traefik
- Access to log statements (see Logs in Figure 6)
- Issue-tracking system information
- Version information, such as Git commit hash, semantic version, and build date and time
- Swagger UI integration (see API Documentation in Figure 6)
- Start and stop buttons to test the resilience of each microservice

To provide these features, PREvant relies on some conventions. For example, to make the logs available, the container must log to standard output, which is a common practice for cloud-native applications [26]. Additionally, PREvant attempts to link the names of the applications to issue tracking information.

To provide version information and Swagger UI integration, PREvant collects information from the microservice itself. Therefore, PREvant employs the web host metadata, proposed in RFC 6415 [6]. If a microservice provides the well-known resource `/.well-known/host-meta.json`, then PREvant can collect the required information and render it in the web interface. Listing 3 illustrates web host metadata of the microservice `shipping` in the application `master`, demonstrating that the OpenAPI specification [8] of the microservice is available at `http://preview.example.com/master/shipping/swagger.json`. When PREvant requests the well-known resource, it provides the HTTP headers `Forwarded` [19] and `X-Forwarded-Prefix` to the microservice, enabling the microservice to generate public links, which are required by the web interface.

Listing 3 Microservices Properties Formulated in Web Host Meta.

```
{
  "properties": {
    "https://schema.org/softwareVersion": "0.9",
    "https://schema.org/dateModified": "2019-08-12T15:31:00Z",
    "https://git-scm.com/docs/git-commit":
       "43de4c6edf3c7ed93cdf8983f1ea7d73115176cc"
  },
  "links": [
    {
      "rel": "https://github.com/OAI/OpenAPI-Specification",
      "href": "http://preview.example.com/master/shipping/swagger.json"
    }
  ]
}
```

5 Development and Quality Assurance Workflows

Aixigo AG utilizes PREvant extensively in daily development activities and has developed substantial workflows that improve the quality assurance of its microservices:

- Aixigo AG utilizes feature branches to prevent the pollution of mainline branches with incomplete features, as discussed in Section 5.1.
- Aixigo's microservices provide flexible configuration capabilities, and PREvant allows changes in configuration set-ups quickly to test different scenarios, as illustrated in Section 5.2.
- Aixigo AG provides bug fixes for older major or minor releases of its microservices. In these cases, PREvant helps to reproduce reported bugs, as outlined in Section 5.3.

Each described workflow occurs before the release to production, because aixigo AG generally cannot access the production area due to legal constraints. Some testing strategies of microservices suggest testing in the production area [15] but this paper does not cover such strategies; rather, it describes workflows that allow developers, domain experts, and sales and team managers to provide improved service quality for in-house developments.

5.1 Feature Branch Based Development

New feature development of a microservice at aixigo AG is subject to strict quality control because several actions have been implemented to ensure high quality: test-driven development [1], code reviews [3], pair programming [27], snapshot and integration tests, end-to-end tests.[10] Additionally, the company has enhanced its development workflow based on reviews by its domain experts.

When a new feature is completed by a developer, such as new functionality in the front-end or a new REST resource, it must be reviewed by a domain expert; the domain experts have

[10] Vocke [25] provides more information about automatic software tests.

experience with REST APIs, so they can judge whether the developer has implemented the feature correctly. When the feature branch of any microservice has been built by the delivery pipeline and the whole application is available on PREvant, as mentioned in Figure 4, the developer notifies the domain expert that the branch is ready for review, and the domain expert performs exploratory tests. If any issues are discovered, then they are reported to the developer; the developer then solves the reported issues.

Even when developers are working on features that are not observable from the outside, PREvant ensures that the application meets quality criteria. For example, aixigo AG's delivery pipelines deploy the feature branches for automatic end-to-end tests (see the acceptance stage). PREvant deploys the whole application and the end-to-end tests are working on the whole application, which ensures that the whole set of microservices is working in conjunction, thus reducing integration time and costs. Due to the quick deployment cycles that PREvant provides, teams at aixigo AG have extended their definitions of done with the clause that the developer is responsible to test the new application features manually (by clicking through the front-end or by utilizing the integrated Swagger UI), which prevents obvious faults by the developer.

When the feature is completed and complies with the definition of done[23, Page 18], the developer merges the feature onto the mainline branch. This invokes a web hook and instructs PREvant to shut down the application, and the development continues with the next feature.

5.2 Configurable and Isolated Environments

Because the microservices of aixigo AG are utilized with different configurations in different production environments, it is crucial that these configurations can be tested in an accessible manner. Therefore, the company utilizes PREvant's REST API to spin up-and-down applications with specific configuration scenarios. Listing 4 illustrates the usage of PREvant's REST API to spin up the application `features.test` with the `order` service in a specific configuration.

Listing 4 REST API Call with Configuration for Test Case.

```
POST /api/apps/features.test HTTP/1.1
Host: preview.example.com
Content-Type: application/json
Accept: application/json

[{
  "serviceName": "order",
  "image": "registry.example.com/a-team/order",
  "files": {
    "/etc/order/conf.d/sales.properties": "some.feature.flag = OFF"
  },
  "env": {
    "FEATURE_FLAG": "ON"
  }
}]
```

In this case the configuration of `order` is influenced by two parameters: the configuration file `/etc/order/conf.d/sales.properties` and the environment variable `FEATURE_FLAG` that toggle features of the service. The configuration of the remaining services has not been changed. When the REST request has been executed, the domain experts and developers can test the service to determine whether it behaves correctly. This feature is crucial to quality assurance when introducing new configuration options because the mainline application runs in a default configuration; otherwise, it would be difficult for domain experts and developers to spin up an application for quality-assurance purposes. PREvant's approach provides additional use cases:

- The microservices with the specific configurations are running in isolation so that domain experts can test the application without disruption from builds and deployments that are executed on the mainline branch. Aixigo AG's domain experts often utilize this feature to clone the mainline branch to test it with the default configuration without disruption. The REST call that is integrated into PREvant's web interface is illustrated in Listing 5. Furthermore, the REST interface provides the query parameter `replicateFrom` to specify which application should be replicated.

Listing 5 REST API Call Cloning Mainline.

```
POST /api/apps/features.test?replicateFrom=master HTTP/1.1
Host: preview.example.com
Content-Type: application/json
Accept: application/json

[]
```

- From time to time, sales managers wish to demonstrate the application to potential customers and must demonstrate the application in a specific configuration. Therefore, they ask aixigo's domain experts to set up an application that can demonstrate feature X or Y.

5.3 Version Picking

Microservices of aixigo AG are running in production in different major or minor versions, and users report bugs in different versions. To reproduce these bugs, domain experts and developers utilize PREvant to spin up an application that runs the specific version with an additional specific configuration of the service. Therefore, a domain expert can utilize the REST API to select a specific container image version, as illustrated in Listing 6.[11]

Listing 6 REST API Call with Version Picking.

```
POST /api/apps/is-it-working-with-v1?replicateFrom=master HTTP/1.1
Host: preview.example.com
Content-Type: application/json
Accept: application/json

[{
  "serviceName": "order",
  "image": "registry.example.com/a-team/order:1.0.2"
}]
```

[11] Sometimes it is useful to compare old and new versions of a microservice in aixigo's sprint reviews; on such occasions, it is convenient to spin it up with PREvant.

PREvant does not provide a user interface for this use case and the feature is only accessible utilizing command line tools or the integrated Swagger UI. However, PREvant's road map contains the extension of the web interface, so that spinning up applications with specific versions and configurations, as illustrated in Listing 4 and 6, is more effective.

6 Companions and Recipes

While PREvant enables workflows that ensure a higher quality of microservices, it also aims to be agnostic to the hosted types of microservices, which means that the type of microservices [10] in development are irrelevant to PREvant. Whether a team develops a function as a service or a self-contained system, the microservices merely need to be packaged as a container image. However, these microservices rely on services that provide some infrastructure, such as databases or OpenID [20] providers, as depicted in the case study in Section 3. To provide these infrastructure services, PREvant offers two types of companions that are infrastructure microservices that are available over the container network while the microservice applications are running. PREvant deploys companions automatically when it received a REST request.

Service Companions Some infrastructure services are specific to a given microservice or are logically owned by a microservice. For example, a database instance such as MariaDB[12] is required by a service, a memory cache such as Memcached[13] is required by another service, and all services require a sidecar proxy.

These infrastructure services must be available for the microservice, and PREvant initiates these services as soon as the dependent microservice spins up.

Application Companions Some infrastructure services must be available for all microservices of an application. For example, Apache Kafka provides a stream-processing platform to establish publish and subscribe messaging between services. Further examples include service discovery providers, API gateways, or OpenID providers.

These types of services are globally available and potentially required by all microservices of the application. Therefore, PREvant initiates these infrastructure services when the application spins up.

To initiate these infrastructure services, PREvant provides configuration options that are illustrated in the following subsections by a set of recipes. These recipes provide some configuration examples that help developers to compose their microservice applications through PREvant. Section 6.1 provides an example of the configuration of database services for all microservices of an application. Additionally, these configuration options provide some templating options so that configurations are dynamically adjusted.

6.1 Service Databases

As stated previously, microservices that are hosted by PREvant often require the provision of a database. Listing 7 illustrates a configuration that ensures initiation of a MariaDB database, which is a service companion.

[12] `https://mariadb.org/`
[13] `http://memcached.org/`

Listing 7 PREvant Configuration: Database Companion.

```
1 [companions.mariadb]
2 type = 'service'
3 image = 'docker.io/library/mariadb:10.3'
4 serviceName = '{{service.name}}-db'
5 env = [
6   'MYSQL_DATABASE={{service.name}}', 'MYSQL_USER={{service.name}}',
    ↪  'MYSQL_PASSWORD={{service.name}}'
7 ]
```

Initially, the companion name must be defined; in this case, the name is `mariadb` (see Line 1). Additionally, the companion type must be defined so that it is valid for every service (see Line 2). Furthermore, the container image as well as the service name must be specified (see Line 3), which results in the DNS name of the database (see Line 4). The configuration of the DNS name can be adjusted by Handlebars templating syntax.[14] In this case, the corresponding microservice name is utilized to derive the DNS name. For example, for the microservice with name `X`, the corresponding database DNS name is `X-db`. Additionally, the environment variables of the database companion are adjusted so that the companion creates a database with a default username and password.

6.2 API Gateway

Some microservice architectures require an API gateway that processes every request before they are forwarded to any microservice. To initiate an API gateway for each application, definitions of the type, image and service name for the companion are required, as illustrated in Listing 8 by `[companions.api-gateway]`.

Listing 8 PREvant Configuration Using An API Gateway.

```
[companions.api-gateway]
type = 'application'
image = 'registry.example.com/a-team/api-gateway:latest'
serviceName = 'api-gateway'

[companions.api-gateway.labels]
'traefik.frontend.rule' = 'PathPrefix:/{{application.name}}/'
'traefik.frontend.priority' = '10000'
```

Furthermore, all requests are routed through the API gateway; therefore, the entry link a domain expert employs to interact with the application is irrelevant. In Listing 8, the configuration section labeled `[companions.api-gateway.labels]` ensures that the default labeling of the API gateway, which is responsible for Traefik's automatic discovery, is overwritten. Here, every request to the application is routed to the API gateway because of the higher priority and the path-prefix rule.[15]

[14] `https://handlebarsjs.com/`

[15] More information about Traefik's configuration options are available at: `https://docs.traefik.io/basics/`

7 Summary

This paper presented the concepts and implementation details of the tool Preview Servant (PREvant), which enables developers and domain experts to perform quality-assurance tasks on their microservice applications. Therefore, PREvant provides a simple REST interface that allows developers to extend their microservices' delivery pipelines with PREvant's composing mechanism (see Section 4), which spins up fully functional applications that can be explored by domain experts. Based on the capabilities of the RESTful interface, aixigo's employees utilize workflows that increase and maintain the microservice quality (see Section 5). Furthermore, PREvant's approach is not tied to these established workflows; rather, it employs an approach that is independent from the microservice application architecture because PREvant enables developers to configure required infrastructure services, as described in Section 6.

Furthermore, PREvant's composition and configuration mechanisms that are integrated in the web interface, allow users to set up dedicated previews in minutes so that sales managers can utilize isolated applications to demonstrate the application to potential customers. PREvant has become a major factor in the quality-assurance process at aixigo AG.

References

1 Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, July 2003.

2 Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, and Manuel Mazzara. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*, 35(3):50–55, May 2018. `doi:10.1109/MS.2018.2141026`.

3 Giuliana Carullo. *Code Reviews 101: The Wisdom of Good Coding*. Giuliana Carullo, May 2019.

4 Lianping Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 32(2):50–54, March 2015. `doi:10.1109/MS.2015.27`.

5 Containous. Traefik: The Cloud Native Edge Router, 2019. Accessed: 2019-08-12. URL: `https://traefik.io/`.

6 B. Cook. Web Host Metadata. Technical report, Internet Engineering Task Force, November 2011. `doi:10.17487/rfc6415`.

7 Cosmin Costache, Octavian Machidon, Adrian Mladin, Florin Sandu, and Razvan Bocu. Software-defined networking of Linux containers. In *RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, pages 1–4, September 2014. `doi:10.1109/RoEduNet-RENAM.2014.6955310`.

8 The Linux Foundation. OpenAPI Initiative, 2019. Accessed: 2019-11-08. URL: `https://www.openapis.org/`.

9 Silvery Fu, Jiangchuan Liu, Xiaowen Chu, and Yueming Hu. Toward a Standard Interface for Cloud Providers: The Container as the Narrow Waist. *IEEE Internet Computing*, 20:66–71, 2016. `doi:10.1109/MIC.2016.25`.

10 Martin Garriga. Towards a Taxonomy of Microservices Architectures. In *Software Engineering and Formal Methods*, pages 203–218. Springer International Publishing, February 2018. `doi:10.1007/978-3-319-74781-1_15`.

11 Anne Marie Glen. [DZone Research] Microservices Priorities and Trends, July 2018. Accessed: 2019-08-12. URL: `https://dzone.com/articles/dzone-research-microservices-priorities-and-trends`.

12 Jesper Holck and Niels Jørgensen. Continuous Integration and Quality Assurance: a case study of two open source projects. *Australasian Journal of Information Systems*, 11(1), November 2003. `doi:10.3127/ajis.v11i1.145`.

13 Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013. doi:10.1109/tcc.2013.10.

14 David Jaramillo, Duy Nguyen, and Robert Smart. Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016*, pages 1–5, March 2016. doi:10.1109/SECON.2016.7506647.

15 Sheroy Marker. Test Strategy for Microservices, May 2018. Accessed: 2019-08-12. URL: https://www.gocd.org/2018/05/08/continuous-delivery-microservices-test-strategy/.

16 Nicholas D. Matsakis and Felix S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104. ACM, 2014. doi:10.1145/2692956.2663188.

17 Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

18 Sam Newman. *Building Microservices.* O'Reilly Media, Inc., 1st edition, 2015.

19 A. Petersson and M. Nilsson. Forwarded HTTP Extension. Technical report, Internet Engineering Task Force, June 2014. doi:10.17487/rfc7239.

20 David Recordon and Drummond Reed. OpenID 2.0: A Platform for User-centric Identity Management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM. doi:10.1145/1179529.1179532.

21 Cesar Saavedra. The State of Microservices Survey 2017 – Eight trends you need to know, December 2017. Accessed: 2019-08-12. URL: https://middlewareblog.redhat.com/2017/12/05/the-state-of-microservices-survey-2017-eight-trends-you-need-to-know/.

22 D. I. Savchenko, Gleb Radchenko, and Ossi Taipale. Microservices validation: Mjolnirr platform case study. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 235–240, May 2015. doi:10.1109/MIPRO.2015.7160271.

23 Jeff Sutherland and Ken Schwaber. The scrum guide. *The definitive guide to scrum: The rules of the game*, 268, 2013. URL: https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf.

24 Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. Microservices in Practice: A Survey Study. In *VEM 2018 - 6th Workshop on Software Visualization, Evolution and Maintenance*, Sao Carlos, Brazil, September 2018. URL: https://hal.inria.fr/hal-01944464.

25 Ham Vocke. The Practical Test Pyramid, February 2018. Accessed: 2019-11-08. URL: https://martinfowler.com/articles/practical-test-pyramid.html.

26 Adam Wiggins. The Twelve-Factor App, 2017. Accessed: 2019-08-12. URL: https://12factor.net/.

27 Laurie Williams. *Pair Programming Illuminated.* Addison-Wesley Professional, July 2002.

28 Eberhard Wolff. *Microservices: Flexible Software Architectures.* CreateSpace Independent Publishing Platform, 2016.

29 Eberhard Wolff. *Microservices: A Practical Guide.* CreateSpace Independent Publishing Platform, April 2018.

30 Olaf Zimmermann. Microservices Tenets. *Comput. Sci.*, 32(3–4):301–310, July 2017. doi:10.1007/s00450-016-0337-0.