Joint Post-proceedings of the Third and Fourth International Conference on Microservices

Microservices 2020, September 2020, Virtual Meeting
Microservices 2022, May 2022, Paris, France

Edited by

Gokila Dorai
Maurizio Gabbrielli
Giulio Manzonetto
Aomar Osmani
Marco Prandini
Gianluigi Zavattaro
Olaf Zimmermann

OASICS

Editors

Gokila Dorai
School of Computer & Cyber Sciences,
Augusta University, Georgia, USA
gdorai@augusta.edu

Maurizio Gabbrielli
University of Bologna, Italy
gabbri@cs.unibo.it

Giulio Manzonetto
Université Paris Cité, CNRS, IRIF,
F-75013, Paris, France
gmanzone@irif.fr

Aomar Osmani
Université Sorbonne Paris Nord, CNRS, LIPN, UMR
7030, Villetaneuse, France
Aomar.Osmani@lipn.univ-paris13.fr

Marco Prandini
University of Bologna, Italy
marco.prandini@unibo.it

Gianluigi Zavattaro
University of Bologna, Italy
gianluigi.zavattaro@unibo.it

Olaf Zimmermann
OST Eastern Switzerland University of
Applied Sciences, Switzerland
olaf.zimmermann@ost.ch

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 978-3-95977-306-5

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/978-3-95977-306-5.

Publication date
December, 2023

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.2020-2022.0
ISBN 978-3-95977-306-5 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs is a series of high-quality conference proceedings across all fields in informatics. 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

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman
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 a European-based non-profit organisation 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 whose aim is to 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 2020, September 2020, Virtual Meeting.

The general theme of Microservices 2020 was the interplay between microservices and cyber security.[2] Wide representation was left open to both scientific papers and application reports regarding different themes. The criteria for acceptance were adjusted towards selecting presentations that could raise interest and spark discussion, rather than seeking complete maturity from a scientific or industrial standpoint.

The program committee consisted of 32 members and the conference received 23 submissions in the form of 2-page abstracts, among which 20 were accepted for presentation. The conference was held online due to Covid-19 restrictions, spanning three days. In addition to the 20 presentations, three keynote speeches were given (by Vaughn Vernon, Sanjiva Weerawarana, and Antonio Brogi), the Microservices Community was introduced, and two plenary sessions were offered merging the audiences of the Bologna Federated Conference on Programming Languages, including the 28th International Workshop on Functional and Logic Programming, the 30th International Symposium on Logic-Based Program Synthesis and Transformation, and the 22nd International Symposium on Principles and Practice of Declarative Programming.

Microservices 2022, May 2022, Paris, France.

The general themes of Microservices 2022 were the aspects of microservices in the age of digital transformation.[3] The program committee consisted of 17 members and the conference received the submission of 20 extended abstracts, among which 16 were accepted for presentation. The contributions covered a broad spectrum of topics related to the following three themes: 1) autonomic microservices and software engineering approaches; 2) architectures and tools; 3) migration to microservices. Each day of the conference was dedicated to one of these themes. The program featured Valentina Lenarduzzi, Fabio Casati and Cesare Pautasso as keynote speakers. On the last day of the conference Christian Walther Bruun held a funding and consortium building workshop, giving insights into the opportunities for obtaining EU funding via the European framework programme Horizon Europe and Eurostars.

---

[1] `https://www.microservices.community`
[2] `https://www.conf-micro.services/2020`
[3] `https://www.conf-micro.services/2022`

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Post-proceedings of Microservices 2020/2022

The present volume compiles contributions from attendees of Microservices 2020 and 2022. The volume received ten submissions of which eigth were accepted for publications after two rounds of peer reviewing. In addition to the contributed papers, this volume includes an invited paper selected by the Program Committee members among the keynote speakers of these two editions of Microservices.

We thank the authors of all submitted proposals for their work in preparing and presenting their contributions. We hope that they benefited from the feedback received during the reviewing process. We also thank the members of the program committees of Microservices edition 2020 and 2022, for their excellent work and enthusiasm. Finally, we want to thank all the donors that provided financial support to the conference and these proceedings: Université Sorbonne Paris Nord, Laboratoire d'Informatique de Paris Nord (LIPN), IUT de Villetaneuse, Philip Morris International, Injenia Srl, Huawei-Edinburgh Joint Lab, and last, but not least, the Microservices community.

Let Industry and Academia meet.

Gokila Dorai
Maurizio Gabbrielli
Giulio Manzonetto
Aomar Osmani
Marco Prandini
Gianluigi Zavattaro
Olaf Zimmermann

Microservices 2020

Microservices 2020 Organisation

General Chair	Maurizio Gabbrielli (University of Bologna)
Program Chairs	Marco Prandini (University of Bologna) Gianluigi Zavattaro (University of Bologna) Olaf Zimmermann (OST Eastern Switzerland University of Applied Sciences)
Local Chairs	Davide Berardi (University of Bologna) Andrea Melis (University of Bologna)
Industrial Liaison Chair	Claudio Guidi (ItalianaSoftware s.r.l.)
Publicity Chair	Marco Peressotti (University of Southern Denmark)
Program Committee	Alessandra Bagnato (SofteamGroup) Davide Berardi (University of Bologna) Simon Bliudze (INRIA) Chiara Bodei (Pisa University) Elisabetta Di Nitto (Politecnico di Milano) Nicola Dragoni (Technical University of Denmark) Saverio Giallorenzo (University of Southern Denmark) Dimka Karastoyanova (University of Groningen) Oliver Kopp (JabRef) Philipp Leitner (University of Gothenburg) Daniel Lübke (Leibniz Universität Hannover) Tiziana Margaria (Lero) Jacopo Mauro (University of Oslo) Balint Maschio (Pixis.co) Andrea Melis (University of Bologna) Rebecca Montanari (University of Bologna) Claus Pahl (Free University of Bozen-Bolzano) Anupama Pathirage (WSO2) Cesare Pautasso (University of Lugano) Marco Peressotti (University of Southern Denmark) Florian Rademacher (University of Applied Sciences and Arts Dortmund) Dakshitha Ratnayake (WSO2) Larisa Safina (Inria - Lille Nord Europe) Maria Seralessandri (ECB) Adina Sirbu (SAP) Jacopo Soldani (University of Pisa) Raffaele Spazzoli (RedHat) Davide Taibi Tampere (University of Technology) Silvia Lizeth Tapia Tarifa (University of Oslo) Massimo Villari (University of Messina)
Keynote Speakers	Antonio Brogi (University of Pisa) : Microservices beyond COVID-19 Vaughn Vernon (Kalel) : Rethinking legacy and monolithic systems Sanjiva Weerawarana (WSO2) : Towards language support for programming microservices

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Microservices 2022

Microservices 2022 Organisation

General Chair	Giulio Manzonetto (University Paris Cité)
Program Chairs	Gokila Dorai (Augusta University) Dimka Karastoyanova (University of Groningen) Aomar Osmani (University Sorbonne Paris Nord)
Local Chair	Damiano Mazza (CNRS, University Sorbonne Paris Nord)
Industrial Liaison Chair	Claudio Guidi (ItalianaSoftware s.r.l.)
Publicity Chairs	Pierre Boudes (University Sorbonne Paris Nord) Florian Rademacher (University of Applied Sciences and Arts Dortmund)
Website Administrator	Jaime Arias (CNRS, University Sorbonne Paris Nord)
Program Committee	Mohamed Abouelsaoud (Cisco) Vasilios Andrikopoulos (University of Groningen) Jaime Arias (CNRS, University Sorbonne Paris Nord) Ibrahim Baggili (University of New Haven) Salima Benbernou (University of Paris) Matteo Bordin (Profesia) Elisabetta Di Nitto (Politecnico di Milano) Lyes Khoukhi (ENSICAEN) Piyush Kumar (Florida State University) Patricia Lago (Vrije Universiteit Amsterdam) Giuseppe Lipari (University of Lille) Anupama Pathirage (WSO2) Pierluigi Plebani (Politecnico di Milano) Dakshitha Ratnayake (WSO2) Maria Seralessandri (European Central Bank (DE) Matteo Zanioli (Alpenite) Asli Zengin (Imola Informatica)
Keynote Speakers	Fabio Casati (Servicenow) : Crossing the chasm between AI services and AI-powered workflows Valentina Lenarduzzi (University of Oulu) : Technical debt and microservices Cesare Pautasso (USI Lugano) : The Microservice Hypothesis

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Scientific Committee of the post-proceedings

Gokila Dorai (Augusta University)
Aomar Osmani (University Sorbonne Paris Nord)
Marco Prandini (University of Bologna)
Gianluigi Zavattaro (University of Bologna)
Olaf Zimmermann (OST Eastern Switzerland University of Applied Sciences, Switzerland)

Subreviewers

Jaime Arias (CNRS, University Sorbonne Paris Nord)
Bradley Boswell (Augusta University)
Saverio Giallorenzo (University of Bologna)
Stefan Kapferer (OST Eastern Switzerland University of Applied Sciences, Switzerland)
Mirko Stocker (OST Eastern Switzerland University of Applied Sciences, Switzerland)

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Microservices Beyond COVID-19

Antonio Brogi
Department of Computer Science, University of Pisa, Italy

Abstract

This article summarises the contents of the invited keynote that I gave back in September 2020 at the "Microservices 2020" Conference, which was held entirely online during the COVID-19 pandemic.

In that keynote, I started from the question of how we can check whether a software application satisfies the main principles of microservices and –if not– of how should we refactor it. To answer that question, I discussed the capacity of existing techniques to automatically extract an architectural description of a microservice-based application, to identify architectural smells possibly violating microservices' principles, and to select suitable refactorings to resolve them. I also discussed how a (minimal) modelling of microservice-based applications can considerably simplify their design and automate their container-based deployment. Finally, I tried to point to some interesting directions for future research on microservices.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases Microservice-based systems

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.1

Category Invited Paper

Funding Work partly funded by UNIPI PRA_2022_64 *"OSMWARE - hOlistic Sustainable Management of distributed softWARE systems"* project, funded by the University of Pisa, Italy.

Acknowledgements I would like to thank all the colleagues with whom I had the pleasure of carrying on the research activities that were described in this keynote, starting from Jacopo Soldani, with whom I shared most of this work, and continuing with (in alphabetical order) Matteo Bogo, Giuseppe Muntoni, Davide Neri, Luca Rinaldi, and Olaf Zimmermann. I would like to thank also Hernan Astudillo, Edoardo Baldini, Javier Berrocal, Giuseppe Bisicchia, Stefano Chessa, Giorgio Dell'Immagine, Stefano Forti, Marco Gaglianese, Juan Luis Herrera, Javad Khalili, Juan M. Murillo, Federica Paganelli, and Francisco Ponce, with whom I co-authored the more recent work (after Microservices 2020) cited in the last part of this article.

1 Design principles, architectural smells and refactorings

I started my keynote by recalling the main motivations and characteristics of microservices, and then I considered the following question:

> *How can architectural smells affecting design principles of microservices be detected and resolved via refactoring?*

Informally speaking, an architectural smell is a "suspect" that the defined architecture may affect a design principle. As an example of possible answer to the above question, I presented the results of the multi-vocal review [8], aimed at identifying the most recognised architectural smells for microservices, and the architectural refactorings to resolve them. That review identified seven architectural smells potentially affecting four design principles of microservices, and 13 refactoring techniques to resolve those architectural smells.

I then presented the μ`Freshener` tool [13], which automatically identifies the architectural smells present in a microservice-based application, and which allows applying architectural refactorings to resolve the identified smells.

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 1; pp. 1:1–1:3
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 From incomplete specifications to running applications

I then moved to considering the question of how to select an appropriate runtime enviroment for each microservice of an application, during the design phase. As an example of possible answer to the above question, I presented the `TosKeriser` tool [3], which automatically completes a TOSCA application specifications by discovering and including Docker-based runtime environments providing the software support needed by each microservice.

I then moved to considering the question of how to suitably package each microservice into the selected runtime environment. As an example of possible answer to the above question, I presented the `TosKose` tool [2], which enables deploying microservice-based applications on top of existing container orchestrators, and to manage each service independently from the container used to run it.

3 Mining the architecture of microservice-based applications

Manually generating the description of the software architecture of an application consisting of dozens, when not hundreds, of microservices is a complex, time-consuming, and error-prone process. Software architects need to be supported by tools capable of automatically mining the software architecture of their microservice-based application.

As an example of such support, I presented the μ`Miner` tool [13], which automatically extracts the software architecture of a " black-box" microservice-based application. Without accessing the application source code, μ`Miner` derives the software architecture from the declarative specification of its Kubernetes deployment, by performing both static and dynamic analyses.

4 Concluding remarks

At the end of the keynote, I summarised the toolchain sketched in Figure 1, obtainined by pipelining the four tools described during the talk.

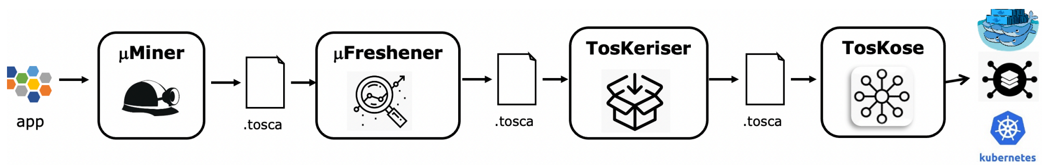


Figure 1 Toolchain example.

Take-home message: The toolchain can be taken as an example of how a (minimal) modelling of microservice-based applications can considerably simplify their design and analysis and allow automating their container-based completion and deployment.

Finally, here is a non-exhaustive list of possible interesting directions for future research on microservices on which I am working with my group and other colleagues:

- improve the techniques for *detecting and resolving architectural smells* in microservice-based applications – with alternative techniques (e.g. like [12]) for automatically extracting the software architecture of an application from its Kubernetes deployment, and for resolving architectural smells by directly modifying the Kubernetes manfiest of an application,

- improve the techniques for *detecting security smells* in microservice-based applications – by identifying the most recognised security smells for microservices (e.g. like in [9]), and by developing automated detectors (e.g. like the extensible `KubeHound` tool [4]),
- improve the tecnhiques for *determining the root causes of microservices' failures* [10] and for *explaining how failures propagate* across microservices (e.g. as in [11]),
- improve the techniques for achieving a *lightweight but effective monitoring* of microservice-based applications deployed on a distributed infrastructure [6],
- consider *sustainaibility* aspects during the entire life-cycle of microservice-based applications [1],
- develop and apply *continuous reasoning techniques* to efficently manage distributed applications in continuity with existing CI/CD pipelines and monitoring tools (e.g. like in [5, 7]).

References

1 Edoardo Baldini, Stefano Chessa, and Antonio Brogi. Estimating the environmental impact of green IoT deployments. *Sensors*, 23(3), 2023. `doi:10.3390/S23031537`.

2 Matteo Bogo, Jacopo Soldani, Davide Neri, and Antonio Brogi. Component-aware Orchestration of Cloud-based Enterprise Applications, from TOSCA to Docker and Kubernetes. *Software: Practice and Experience*, 50:1793–1821, 2020. `doi:10.1002/SPE.2848`.

3 Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. Orchestrating incomplete TOSCA applications with Docker. *Science of Computer Programming*, 166:194–213, 2018. `doi:10.1016/J.SCICO.2018.07.005`.

4 Giorgio Dell'Immagine, Jacopo Soldani, and Antonio Brogi. KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments. *Future Internet*, 15(7), 2023. `doi:10.3390/FI15070228`.

5 Stefano Forti, Giuseppe Bisicchia, and Antonio Brogi. Declarative Continuous Reasoning in the Cloud-IoT Continuum. *Journal of Logic and Computation*, 32(2):206–232, 2022. `doi:10.1093/LOGCOM/EXAB083`.

6 Marco Gaglianese, Stefano Forti, Federica Paganelli, and Antonio Brogi. Assessing and enhancing a Cloud-IoT monitoring service over federated testbeds. *Future Generation Computer Systems*, 147:77–92, 2023. `doi:10.1016/J.FUTURE.2023.04.026`.

7 Juan Luis Herrera, Javier Berrocal, Stefano Forti, Antonio Brogi, and Juan M. Murillo. Continuous QoS-Aware Adaptation of Cloud-IoT Application Placements. *Computing*, 105:2037–2059, 2023. `doi:10.1007/S00607-023-01153-1`.

8 Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. Design principles, architectural smells and refactorings for microservices: A multivocal review. *Software-Intensive Cyber-Physical Systems*, 35:3–15, 2020. `doi:10.1007/S00450-019-00407-8`.

9 Francisco Ponce, Jacopo Soldani, Hernan Astudillo, and Antonio Brogi. Smells and Refactorings for Microservices Security: A Multivocal Literature Review. *Journal of Systems & Software*, 4(C), 2023. `doi:10.1016/J.JSS.2022.111393`.

10 Jacopo Soldani and Antonio Brogi. Anomaly Detection and Failure Root Cause Analysis in (Micro)Service-Based Cloud Applications. *ACM Computing Surveys*, 55(3):1–39, 2022. `doi:10.1145/3501297`.

11 Jacopo Soldani, Stefano Forti, and Antonio Brogi. yRCA: An explainable failure root cause analyser. *Science of Computer Programming*, 230, 2023. `doi:10.1016/J.SCICO.2023.102997`.

12 Jacopo Soldani, Javad Khalili, and Antonio Brogi. Offline Mining of Microservice-Based Architectures. *SN Computer Science*, 4, 2023. `doi:10.1007/S42979-023-01721-4`.

13 Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621, 2021. `doi:10.1002/SPE.2974`.

Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements

Tomas Cerny
Systems and Industrial Engineering, University of Arizona, Tucson, AZ, USA

Davide Taibi
Oulu University, Finland

Abstract

Microservice architecture is the mainstream to fuel cloud-native systems with small service sets developed and deployed independently. The independent nature of this modular architecture also leads to challenges and gaps practitioners did not face in system monoliths. One of the major challenges with decentralization and its independent microservices that are managed by separate teams is that the evolving system architecture easily deviates far from the original plans, and it becomes difficult to maintain. Literature often refers to this process as system architecture degradation. Especially in the context of microservices, available tools are limited. This article challenges the audience on how static analysis could contribute to microservice system development and management, particularly managing architectural degradation. It elaborates on challenges and needed changes in the traditional code analysis to better fit these systems. Consequently, it discusses implications for practitioners once robust static analysis tools become available.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Computer systems organization → Cloud computing; Software and its engineering → Automated static analysis

Keywords and phrases Microservice Architecture, Static Analysis, Reasoning, Decentralization

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.2

Funding *Tomas Cerny*: This material is based upon work supported by the National Science Foundation under Grant No. 2245287, and grant from Red Hat Research `https://research.redhat.com`.
Davide Taibi: This material is based upon work supported by the ADOMS Grant from Ulla Tuominen Foundation and a grant from the Academy of Finland (grant n. 349488 - MuFAno).

1 Introduction

Cloud-native systems are designed to take full advantage of the cloud infrastructure. The 12-factor app methodology [46] provides guidance on designing, building and deploying these systems. For instance, cloud-native systems have, as the foundation, the microservice architecture, utilize containers, and follow Continuous Integration and Delivery (CD/CI).

The microservice architectural style is a paradigm for developing systems as a suite of small, self-contained, and autonomous services communicating through a lightweight protocol. Each microservice has its own codebase with a separate configuration to facilitate its individual decentralized evolution. This, as a result, enables the separation of duties for roles like architects, developers, and DevOps. It also aligns well with Conways's law, stating that organizations should design systems to mirror their own communication structure.

While functionality gets well divided with microservices, overlaps across individual microservice' still exist. Each microservice has a bounded context within the overall system, and still, since microservices interact, the overlap is inevitable. Implications from these overlaps are reflected in the microservice codebase. Since the microservice codebase remains

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 2; pp. 2:1–2:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

self-contained, overlaps mean partially restated definitions, typically re-implemented in a particular framework version. This restatement can relate to data definitions of processed information, encapsulated knowledge, business logic, or other enforcement related to various policies (i.e., security, constraints, privacy, etc.). However, this overlap is uncontrolled throughout the system evolution, and there are fragile mechanisms to assess consistency errors. Thus, once any of these definitions change in the microservice codebase, there is no direct indication of the definition being restated elsewhere in other codebases.

In addition to functional decomposition, one would expect microservices to cope with the separation of concerns. While there are the same means as in any other component-based development, decentralization leads to the scattering of different concerns, lacking a single focal point. Such concern separation might be well-managed on a single codebase level, but it might get lost with the decentralization and the existence of multiple codebases. While infrastructure like centralized configuration servers and API gateway exists, i.e., to enable telemetry and tracing, these cannot be misused beyond their original purpose, leading to anti-patterns.

We typically aim to separate concerns in software systems to provide better readability and maintainability [29]. We can do a micro-management and design solid concern separation per each microservice. However, this would only relate to a single microservice, not the whole system. The question is whether we need to see a certain concern from the entire system perspective. Suppose we are architects; most likely, the answer is yes. For instance, to focus on system privacy concerns. To make informed decisions, developers must see dependent (i.e., interacting) microservices aligned across selected concerns. We must keep in mind with the self-contained microservice nature and the decentralized system perspective, each concern of a certain type is re-defined and encapsulated across microservices. This might be one of the greatest disappointments when migrating from monolith systems. As an example, the consequence of security assessment is that each microservice has to be analyzed individually. Then, the extracted knowledge must be combined ad-hoc, which is tedious, time-consuming, error-prone, and does not scale with agile development. Unfortunately, with microservice architecture, we must accept that the system must deal with "*scattered concerns*" [9].

In this article, we discuss how static analysis could contribute to solving the shortcomings of microservices-based systems. We emphasize how future tools should adapt to better fit these systems' specifics. We base our discussion on case studies and prototype tools we developed with our research teams.

In the remainder of this paper, we discuss the current approaches to assess cloud-native systems (Section 2). Next, Section 3 focuses on changes to static analysis tools to better align with cloud-native. Finally, Section 4 discusses the implications and impact on involved stakeholders once these tools become robust and available, while Section 5 concludes the paper.

2 Current Trends

Researchers often resort to applying dynamic system analysis to address various microservice challenges. The dynamic analysis can undoubtedly uncover service dependency graphs and bring them a more centric system view.

However, to uncover these artifacts, we need to invest in different efforts. First, the uncovered artifacts can only be as complete as the underlying systems tests or system interaction. This means that we could extract a complete graph if we had complete test coverage [18, 38]. However, complete test coverage is expensive, and it must adapt to system evolution. Alternatively, we could use production system traffic, but even then, there is no guarantee of complete system coverage. Second, we do not want customers to identify, i.e., cyclic dependency in production, and should target system analysis before it ships to production. The dynamic analysis will need the system to run to perform interaction to uncover the previously mentioned artifacts, which is time-consuming. We would need a lot of computational power if we anticipate analyzing the system for every new code change (commit) in the codebase. It also takes time to perform the tests (especially with full coverage).

Dynamic analysis can be performed by system monitoring (i.e., with telemetry `https://opentelemetry.io`) or by centralized log tracing. Traces are produced through logging augmented with correlation identifiers, and log statements can represent what developers added to the system or instrumented to important system components (i.e., endpoints). The great advantage of this approach is its platform agnosticism.

However, one must recognize the necessity of additional extensions to microservices, and their infrastructure to integrate centralized logging and tracing [8]. For instance, correlation identifiers must be introduced, log centralization must be in place, and health checks must be provided for the most advanced reporting. The dynamic analysis led by telemetry can determine microservice dependencies from call-graphs [19, 43, 31], a heat map of how often are certain endpoints reached.

Nevertheless, the dynamic analysis has limitations. It cannot access details exclusive to codebases (such as which component is responsible for a given endpoint business logic, etc.) [18]. We must also consider the separation of duty relevant to telemetry. Different roles might have different needs. Developers might want to know if their change did not break microservice neighbors. However, DevOps manages telemetry or centralized logs with tracing, not Developers [4]. Such role division introduces indirection in reporting, multi-step interpretation, and latency between what has been developed and what has been identified.

The metaphor for dynamic and static analysis could be whether to use typed-safe or interpreted languages with no type-safety. Developers who manage an individual microservice codebase likely take advantage of quick code change checks. These are based on static analysis and are often part of integrated development environments, build files, or added to the CD/CI pipelines. However, the limit of these tools today is that they only relate to a single codebase. The emerging challenge is that successful new tools will need to operate across codebases and combine results with seeing the system as a whole rather than as separate pieces of the greater puzzle [21, 34].

When comparing static and dynamic analysis, we must understand that these instruments have two different targets. One can inform about the underlying structures and the white-box view; the other details how the system operates within a black-box view.

Naturally, there are overlaps. Both approaches can identify the system's endpoints or its microservices [18]. However, it is also the boundary of where the approach limits stand. Anything below endpoints is the goodwill of tracing instrumentation to access it in dynamic analysis [7]. However, there is a toll since code instrumentation to add additional logging has a performance impact. On the other hand, anything below endpoints is a native perspective for static analysis. On the contrary, dynamic analysis will reveal how users use the system and endpoints, which are more popular than others, etc.

We can observe that static analysis is rather in the control of developers, and dynamic analysis is more relevant to operations (i.e., DevOps engineers). Still, for other stakeholders, i.e., to perform a security assessment, we might need a combination of both. Ideally, both perspectives combine symbiotically, giving comprehensive system insights into its dynamics.

Still, none of the static or dynamic analyses could explain how developers organize or how the system changes over time. To answer such a perspective, Mining Software Repositories (MSR) can indicate how the system structure changes over time and does the organization around particular microservices changes. We can collect additional information related to version control messages, possibly linked to issues in ticketing systems. MSR often time connects with static analysis.

The primary input for static analysis is the system code (source, bytecode, or binary) [2]. In the most basic way, source code is parsed into an Abstract-Syntax Tree (AST) and then converted to other forms of graphs. These phrases are then traversed to perform defined verification or match various anti-patterns [44, 13]. The result of such parsing typically generates an intermediate representation (IR) or a model of the system in which the extracted information and structures are reasoned about.

Static analysis does not only consume code or code changes pushed by MSR. The cloud-native design typically involves build files and container configuration files in the repository, and these files can be easily analyzed to help determine topology [22, 39, 27] and involved technology and future static analysis approaches cannot omit these aspects.

3 Conventional Static Analysis versus Microservice Systems

As introduced previously, conventional static analysis performs on a single codebase. It determines dependencies across various internal structures using an abstraction that makes reasoning about a given system easier [2]. However, cloud-native systems are decentralized, possibly with a self-contained codebase per microservice. This difference makes it more challenging to deliver anticipated results to understand the system's dependencies or reason holistically since each codebase could employ a different framework, platform, or library version. As a result, it is necessary to consider static analysis per each codebase.

Multi-codebase is not the only challenge; individual analysis results do not combine linearly next to each other. Instead, they need careful interweaving in the scope where they overlap - across bounded contexts and interaction. By recognizing these connections, new tools could derive a virtual holistic perspective of the overall system with fine granularity of inter-microservice dependencies.

A good tactic is necessary to overcome the above challenges in the context of polyglot systems. Since many platforms can be used, it is unavoidable to employ multiple platform parsers. The result of all such efforts should be in the form of a unified intermediate representation. This will also enable intermediate representation interweaving that does not need to deal with microservice platform heterogeneity.

In our research and prototyping[45, 6][1] and [28][2], we focused on microservice middleware, and the detection communication patterns between services [35, 42, 41] and on metrics to detect coupling based on the interaction between microservices [1] detected with static analysis [33].

Furthermore, we observed that most microservices would be developed using particular platform frameworks that introduce components [14, 37]. For example, consider Spring, Java Enterprise, C#, and Django. Even if components would not be employed, a good programming convention would be established following separating concerns on the codebase level. With a focus on such practice, we determined that low-level code analysis might

[1] `https://cloudhubs.ecs.baylor.edu/prophet/`
[2] MicroDepGraph `https://github.com/clowee/MicroDepGraph`

Table 1 Static analysis vs. Microservices: Cultural clash.

	Conventional static analysis	Microservice traits
Code perspective	Plain 'low-level' code expected with low-level intermediate representation	Enterprise standards and components are the main constructs
Codebase	Limited to a single codebase; linear result combination does not work for microservices given their dependencies	Decentralized with decentralized codebases per microservice
Heterogeneity	Language-specific (monoglot)	Heterogeneous system parts (polyglots)

be unnecessary, but still, this is the conventional approach. Instead, one should focus on components like data entities, repositories, services, and controllers. In addition, the internal call-graphs across components and involved high-level structures should be detected (i.e., remote-procedure calls, REST-call, event registration, etc.). The result would be in the form of an intermediate representation that forms a component call-graph.

We summarize the gaps for conventional static analysis when placed into a contract with microservices in Table 1. We propose that microservice-aware static analysis may operate with polyglot systems built with heterogeneous platforms; it must recognize high-level structures and components and properly combine results across analyzed codebases.

4 Proposed Methodology for Microservice-aware Static Analysis

The key decision for static analysis is to choose the proper system intermediate representation. We chose a component call-graph since many platforms use components.

With an emphasis on operating with an intermediate representation that forms a component call-graph, we consider the utility of conventional code parsers to determine AST to detect the system structure. Based on common component types across different frameworks, it is possible to detect components, their properties, specifics, and connections. However, this often drags the approach to become platform-specific in its nature.

Nevertheless, working with AST or its converted graphs like control-flow graphs enables us to determine the anticipated intermediate representation of the component call-graph per each microservice.

We have initially assessed this approach on Spring and Java Enterprise platforms on two system benchmarks [47, 10] with success.

In our follow-up work [37][3], we intended to generalize the process across platforms. As a result, we proposed that the AST be extended to be a superset across multiple languages, which leads to a Language-Agnostic AST (LAAST). Some rules can be added to convert constructs across platforms (i.e., defer operator or switch into if/else).

Using LAAST, it is fairly simple to build or customize pattern-matching agents to detect components or higher-level structures. Thus, a common set can be established for conventional framework components. Still, the developer can customize these matches for naming conventions and apply custom callback to populate the component call-graph intermediate representation with a given component properties.

[3] `https://github.com/cloudhubs/source-code-parser`

We have tested this follow-up prototype with success on the previous testbeds [47, 10] and on C++ [20], manually validating precision and recall for component detection above 95% [37].

With the component call-graph intermediate representation of each microservice, we can consider interweaving them. The bottom-up approach is to join involved data models in given bounded contexts. The horizontal approach is to predict possible inter-service communication. This can be accomplished by detecting remote calls to certain endpoints, identifying relative paths, HTTP types, and parameters, and matching them to endpoint signatures. We recognize that only dynamic analysis can recognize inter-service communication with perfect precision. Still, if we solely consider static analysis, this results in the best approximation, and static analysis is about approximation. However, other interactions based on events and brokers can also be considered.

To interweave microservices, first, overlaps with data entities should be identified. Using LAAST matching agents, we can identify entities and reason about their interconnections to derive a data model per each bounded context. For instance, we can use similarity from natural-language processing, Wu-Palmer algorithm [23] to determine potential matches in entities across microservice intermediate representations. Placing identified entities in an overlay helps us connect intermediate representations together and build a context map.

However, other techniques can be used. The next strategy targets cross-service interaction. We consider possible remote calls between microservices. Similarly, we operate on LAAST to identify endpoints, relative paths, HTTP types and parameters, and similarly remote calls within services, which we match based on HTTP types, relative paths, and parameters. We can determine additional dependencies across microservices that strengthen the previously assembled overlay with this route. Performing this across all microservices, we determine the holistic system component call-graph intermediate representation, which corresponds to the latest state of the system.

In addition to the above, we can also account for configuration files in the codebase. This is especially relevant to container descriptors that are part of cloud-native codebases.

We have tested the proposed interweaving on the previously mentioned testbeds [47, 10, 20] and assessed the results manually, with few associations missing in the resulting context map and few unidentified connections in the inter-service interaction [6]. The missing connections were all due to ambiguity caused by choosing from multiple potential URLs at endpoints, which we did not optimize our prototype for, expecting each endpoint to match a single URL.

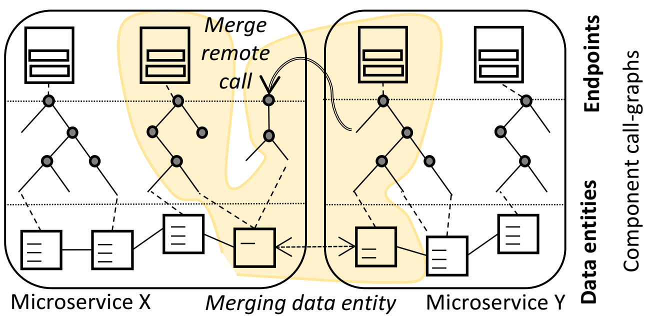


Figure 1 Example interweaving of two microservices X and Y based on remote calls and similar data entities.

Table 2 Microservices-aware static analysis: procedural steps.

Stage	Microservice-aware static analysis	Realization
Step 1	Recognize components in code along with high-level constructs (i.e., remote calls, component interaction)	Use AST or other graphs
Step 2	Establish microservice intermediate representations	Consider component call-graph
Step 3 (optional)	Unification across platforms	Consider language agnostic AST
Step 4	Connect individual microservice intermediate representations (see Figure 1)	• Data entity overlaps • Inter-service calls (remote calls) • Use container descriptors

To summarize the methodology process, we highlight important steps in Table 2. We also illustrate two mentioned interweaving techniques in the context of component call-graphs from two different microservices. This is captured in Figure 1.

5 Discussion on Implications and Challenges

The primary motivation behind the static analysis is automated reasoning and reports [2], as well as system architecture reconstruction [45]. With the ability to operate across the holistic system or multiple microservices, developers (as opposed to DevOps) could gain new aid to quickly understand the impact of their changes [3]. Or get quick feedback on newly introduced anti-patterns [13, 44] and lowered quality metrics. Table 3 lists selected challenge areas.

Considering different concerns, one could be assessing whether the system complies with various organizational policies. Currently, analysts need to review the codebase to determine compliance. Having a holistic system intermediate representation might become easier to assess. Similarly to consistency checking, certain policies could be evaluated.

One related venue for discussion and research is the consistency of business logic. Analyzing business logic is difficult from code, even though we know that service components are to be encapsulated and we can track control and data flows. This opens the question of consistency checking across microservices, which is certainly attractive. However, this also has to consider that modern frameworks add rules via method interception. For instance, frameworks like Drools can greatly reduce management efforts for business rules; however, these again apply to a single codebase. Integrating a configuration server in cloud-native methodology could open a non-intrusive path for centralizing such rules that are now scattered across the system. It would simply utilize principles of generative programming to accomplish this.

Considering the above problem areas (business logic consistency, policies) or other examples like security and privacy, the root cause of problems and dependencies is the manifestation of scattered concerns. Static analysis can extract information about selected concerns from each microservice and put them next to each other to centralize the perspective. This can be accomplished, for instance, by using the component call-graph as the system representation and augmenting its perspectives related to given concerns by targeted code analysis.

Table 3 Selected of challenge areas for static analysis in microservices.

Automated vs. human expert reasoning – anti-patterns, metrics, reports, etc.
Scattered concerns – compliance and consistency in policies, business logic, privacy, security, etc.
Formal methods/verification – using information reconstructed from the code.
Software Architecture Reconstruction – how to perform, which sources to include.
Holistic perspective – how to derive it, which perspectives to consider.
Human-centered perspective – what system information to display in given perspectives to support expert reasoning about the system.
System aspect visualization – augmented/virtual reality, 3D models, etc.
System evolution assessment – many self-contained moving parts must be considered.
Evolution modeling – conversion of intermediate representations system models with instruments designed to speculate on evolution.

Another important avenue for research is the derivation of the system's architectural perspectives that would centralize concerns that are now scattered in decentralized codebases of microservices. For instance, architects might have the motivation to analyze the system's context map or canonical data model, or analysts need to have a single focal point for security assessment and understand all business constraints applied in the system. All these perspectives are necessary to make informed decisions reflecting the current system state, which is currently very difficult to obtain from cloud-native systems.

Results from the static analysis can be accompanied by models for formal verification to aid with trade-off analysis, system evolution planning, or behavior prediction to guarantee correctness. For instance, static analysis can extract a system's intermediate representation and convert it to an architectural description language [30] to help human experts speculatively extend the system via abstract models. Abstractions could also involve choreographic programming to ensure correct concurrency [15, 16]

Despite the above details related to reasoning, the human-centered view should exist and consider visual representation to properly articulate and interpret various concerns, but human experts [3]. In such views, experts could determine anticipated consistency and sketch the system's architectural perspective. Much work has been done in this context, recognizing that architecture can be described through various views [32]. The process is known as Software Architecture Reconstruction (SAR) [36]. As demonstrated in previous works, it can be accomplished through static analysis [45]. Still, researchers address this for microservices through tedious manual code review, possibly outdated documentation assessment [36], or dynamic analysis, which provides only a subset of information, structure, and detail to what is necessary. Software architecture reconstruction fits well with the static analysis process we experimented with.

The typical output of software architecture reconstruction is a system model for answering questions or just reasoning. Reasoning can be automated, such as consistency violation detection or anti-pattern detection [13, 44]. Alternatively, we can also combine these to visually represent certain systems' architectural viewpoints [26].

With regard to system evolution, if we track service interconnection that disappears with an update, something might be wrong with the update, and the developer might be notified about such change impact. This could greatly improve conformance/consistency checking across microservice evolution, which is currently very fragile due to horizontal separation of duty where distinct development teams manage different microservice codebases. However, specific strategies to do so remain to be addressed.

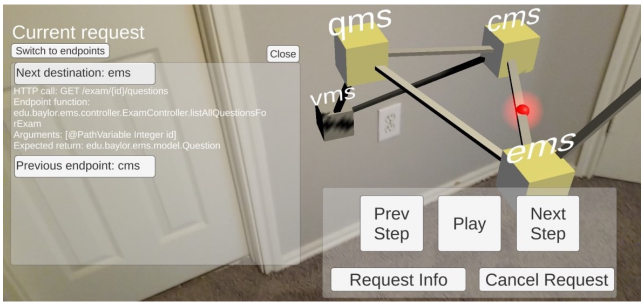


Figure 2 Our AR prototype for SAR and communication simulation.

In the context of challenges for microservices, a broad study by Borgner et al. [5] identified a large set of additional problems that practitioners face with microservices. There is a promising potential for microservice-aware static analysis to address these problems. In the study, the most frequently mentioned issues were missing system-centric perspectives, inter-service dependencies, coordination between decentralized teams, and challenges with outdated documentation. They also mention challenges related to microservice integration, API-breaking changes, etc. We strongly believe all these challenges could be leveraged by introducing robust static analysis tools for cloud-native systems.

6 Experimental Evaluation

We have implemented our methodology in various prototype tools and assessed benefits, limitations, and implications from the Microservice-aware static analysis in cloud-native systems with broader detail.

For instance, we have approached software architecture reconstruction in microservice systems [6, 45] and managed to derive four architectural viewpoints that present the decentralized system as if it was a virtual monolith. This has the potential to realign current static analysis tools to operate on the holistic system.

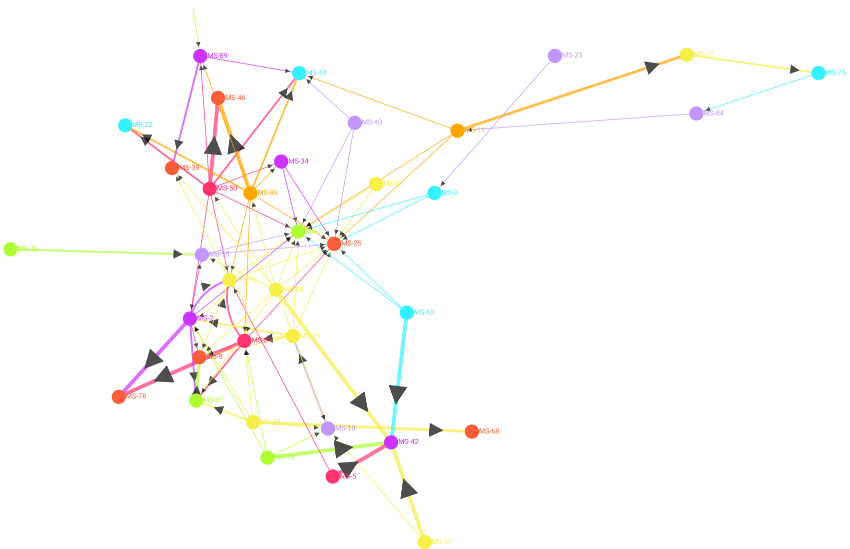


Figure 3 Our interactive two-dimensional service dependency graph visualization.

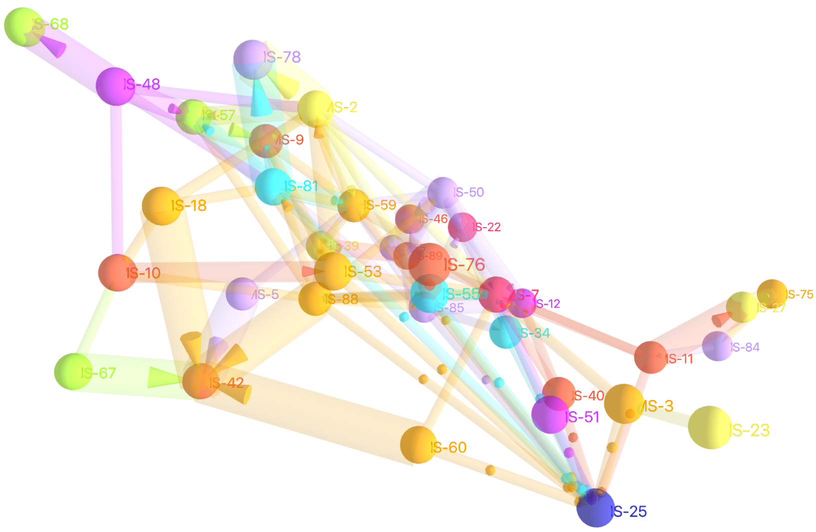


Figure 4 Our interactive three-dimensional service dependency graph visualization.

In addition, we used our model for automated reasoning to detect access policy consistency errors [17] across endpoints. For instance, we might consider access rights from a single microservice context, but when they interplay, they might consider different access rights leading to inconsistencies and possibly vulnerability.

Furthermore, we used our component call-graph intermediate representation along with the architectural viewpoints to detect eleven microservice-specific bad smells [44].

Moreover, the component call-graph intermediate representation has proven well-suited as a model for semantic clone detection across system endpoints [40], which is important when different teams reinvent the same functionality, not knowing about co-existing endpoints.

The granularity of components is suitable for developers in development frameworks. It fits well with the granularity of graph nodes, which makes the visual connection between models and code easy to comprehend.

In addition, we have also researched visualization of systems architecture based on microservices [6]. We have considered that established visualization techniques are too "static" when it comes to developer needs. Thus, we focused on interactivity. Furthermore, we proposed that conventional techniques visualize and model system architecture needs to be reconsidered given the space constraints needed by microservices [11]. Large or even medium-size microservices systems rendered in conventional visualizations require too much space to display relevant information. Thus, in addition to interactivity, we considered three-dimensional spatial visualization.

We share our proof of concepts called Microvision [12, 11] in Figure 2 highlighting one of the systems testbeds. In a large user study involving experts and novices [3], we identified that the benefits of three-dimensional spatial visualization come from mid-size systems and enable novices to identify architectural properties and anomalies as quickly as experts.

However, we also started investigating more web-friendly models that can render in two and three dimensions [24]. In the context of anti-pattern or smell detection, we also considered their visualization in these models [25]. Figures 3 and 4 illustrate our new visual models for two and three-dimensional service dependency graphs, and Figure 5 previews a cyclic dependency anti-pattern highlight in the service dependency graph [26].

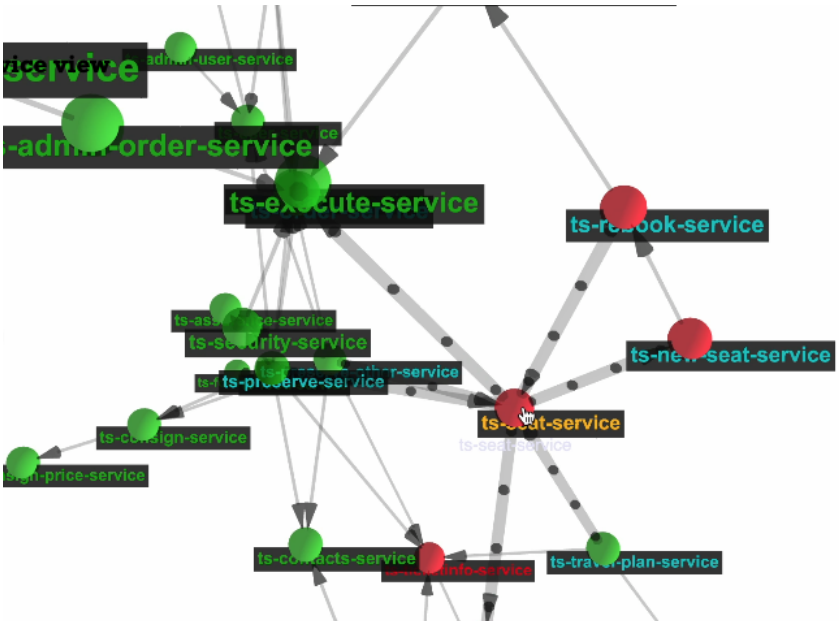


Figure 5 Highlighting detected anti-patterns in the visualization.

In summary, microservice-aware static analysis has great potential to address current gaps and challenges with microservices. This article does not anticipate that static analysis is superior to dynamic analysis. It is meant for different goals and challenges; clearly, a broad research opportunity exists for combined analysis.

7 Conclusion

This article discusses static analysis in the context of microservices and cloud-native design. While static analysis is recognized for many benefits, it has not been widely adopted and used for challenges faced in cloud-native system development. We have listed major obstacles preventing static analysis from operating on the holistic system. We point to our experiments that attempted to interweave intermediate representations of microservices to enable such operation. We believe the scientific and industrial community should put more effort into developing robust tools to help developers better face system evolution and maintenance tasks. In future work, we plan to continue our research in this direction of combining decentralized systems. We will also continue to develop prototypes across languages, demonstrating the ability to assess heterogeneous systems to provide a single focal point when assessing certain information and concerns.

References

1 Amr S Abdelfattah and Tomas Cerny. The microservice dependency matrix. *CoRR*, abs/2309.02804, 2023. `doi:10.48550/ARXIV.2309.02804`.

2 Amr S. Abdelfattah and Tomas Cerny. Roadmap to reasoning in microservice systems: A rapid review. *Applied Sciences*, 13(3), 2023.

3 Amr S Abdelfattah, Tomas Cerny, Davide Taibi, and Sira Vegas. Comparing 2d and augmented reality visualizations for microservice system understandability: A controlled experiment. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 135–145, 2023. doi:10.1109/ICPC58990.2023.00028.

4 Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. Using microservice telemetry data for system dynamic analysis. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 29–38. IEEE, 2022. doi:10.1109/SOSE55356.2022.00010.

5 J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104, 2021. doi:10.1007/S10664-021-09999-9.

6 V. Bushong, D. Das, and T. Cerny. Reconstructing the holistic architecture of microservice systems using static analysis. In *Int. Conf. on Cloud Computing and Services Science (CLOSER)*, 2022. doi:10.5220/0011032100003200.

7 Vincent Bushong, Diptal Das, and Tomas Cerny. Reconstructing the holistic architecture of microservice systems using static analysis. In *Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER*, pages 149–157, 2022. doi:10.5220/0011032100003200.

8 John Carnell and Illary Huaylupo Sánchez. *Spring microservices in action.* Manning Publications Co., 2nd ed. Shelter Island, NY, USA, 2021. URL: https://www.manning.com/books/spring-microservices-in-action-second-edition.

9 Tomas Cerny. Aspect-oriented challenges in system integration with microservices, soa and iot. *Enterprise Information Systems*, 13(4):467–489, 2019. doi:10.1080/17517575.2018.1462406.

10 Tomas Cerny. Microservice Testbed for Texas Teacher Examination, 2020. https://github.com/cloudhubs/tms2020, last accessed 1/2/2022. URL: https://github.com/cloudhubs/tms2020.

11 Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microservice architecture reconstruction and visualization techniques: A review. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 39–48, 2022. doi:10.1109/SOSE55356.2022.00011.

12 Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 49–58, 2022. doi:10.1109/SOSE55356.2022.00012.

13 Tomas Cerny, Amr S. Abdelfattah, Abdullah Al Maruf, Andrea Janes, and Davide Taibi. Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, page 111829, 2023. doi:10.1016/J.JSS.2023.111829.

14 Tomas Cerny, Jan Svacina, Dipta Das, Vincent Bushong, Miroslav Bures, Pavel Tisnovsky, Karel Frajtak, Dongwan Shin, and Jun Huang. On code analysis opportunities and challenges for enterprise systems and microservices. *IEEE Access*, 8:159449–159470, 2020. doi:10.1109/ACCESS.2020.3019985.

15 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In *Formal Techniques for Distributed Objects, Components, and Systems: 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings 37*, pages 92–107. Springer, 2017. doi:10.1007/978-3-319-60225-7_7.

16 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.

17 D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas. On automated rbac assessment by constructing a centralized perspective for microservice mesh. *PeerJ Computer Science*, 7:e376, 2021. doi:10.7717/PEERJ-CS.376.

18 Amr Elsayed, Tomas Cerny, Jorge Yero Salazar, Austin Lehman, Joshua Hunter, Ashley Bickham, and Davide Taibi. End-to-end test coverage metrics in microservice systems: An automated approach. *CoRR*, abs/2308.09257, 2023. doi:10.48550/ARXIV.2308.09257.

19 Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *Queue*, 16(4):10:44–10:65, aug 2018. doi:10.1145/3277539.3277541.

20 Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019. doi:10.1145/3297858.3304013.

21 Mia E Gortney, Patrick E Harris, Tomas Cerny, Abdullah Al Maruf, Miroslav Bures, Davide Taibi, and Pavel Tisnovsky. Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. *IEEE Access*, 2022. doi:10.1109/ACCESS.2022.3221130.

22 G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53, 2017. doi:10.1109/ICSAW.2017.48.

23 L. Han, A. L. Kashyap, T. Finin, J. Mayfield, and J. Weese. UMBC_EBIQUITY-CORE: Semantic textual similarity systems. In *Conf. on Lexical and Computational Semantics*, 2013. URL: https://aclanthology.org/S13-1005/.

24 P. Harris, M. Gortney, A.S. Abdelfattah, and Tomas Cerny. Designing a system-centered view to microservices using service dependency graphs: Elaborating on 2d and 3d visualization. In *The Recent Advances in Transdisciplinary Data Science: First Southwest Data Science Conference, SDSC 2022, Waco, TX, USA, March 25–26, 2022, Revised Selected Papers*, 2023.

25 A. Huizinga, G. Parker, A.S. Abdelfattah, X Li, T. Cerny, and D. Taibi. Detecting microservice anti-patterns using interactive service call graphs: Effort assessment. In *The Recent Advances in Transdisciplinary Data Science: First Southwest Data Science Conference, SDSC 2022, Waco, TX, USA, March 25–26, 2022, Revised Selected Papers*, 2023.

26 Austin Huizinga, Hossain Chy, Md Showkat, Harris Patrick, Parker Garrett, Mia Gortney, Cerny Tomas, Dario Amoroso d'Aragona, and Taibi Davide. Microprospect: Visualizing microservice system architecture evolution and anti-patterns. In *Software Architecture. ECSA 2023 Tracks and Workshops*, 2023.

27 A. Ibrahim, S. Bozhinoski, and A. Pretschner. Attack graph generation for microservice architecture. In *Symposium on Applied Computing*, 2019. doi:10.1145/3297280.3297401.

28 Mohammad Rahman Imranur, Sebastiano Panichella, and Davide Taibi. A curated dataset of microservices-based systems. In *Joint Proceedings of the Summer School on Software Maintenance and Evolution*. CEUR-WS, sep 2019. URL: http://arxiv.org/abs/1909.03249.

29 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*, pages 220–242. Springer, 1997. doi:10.1007/BFB0053381.

30 Luka Lelovic, Michael Mathews, Amr Abdelfattah, and Tomas Cerny. Microservices architecture language for describing service view. In *13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, 2023. doi:10.5220/0011850200003488.

31 S. Ma, C. Fan, Y. Chuang, W. Lee, S. Lee, and N. Hsueh. Using service dependency graph to analyze and test microservices. In *42nd Annual Computer Software and Applications Conf.*, 2018. doi:10.1109/COMPSAC.2018.10207.

32 Liam O'Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, Carnegie Mellon University, jan 2002. doi:10.1184/R1/6583982.v1.

33 Sebastiano Panichella, Mohammad Rahman Imranur, and Davide Taibi. Structural coupling for microservices. In *11th International Conference on Cloud Computing and Services Science*, apr 2021. doi:10.5220/0010481902800287.

34 Garrett Parker, Samuel Kim, Abdullah Al Maruf, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, and Davide Taibi. Visualizing anti-patterns in microservices at runtime: A systematic mapping study. *IEEE Access*, 2023. doi:10.1109/ACCESS.2023.3236165.

35 Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, pages 92–97, New York, NY, USA, 2020. doi:10.1145/3387906.3388625.

36 F. Rademacher, S. Sachweh, and A. Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*. Springer International Publishing, 2020. doi:10.1007/978-3-030-49418-6_21.

37 Micah Schiewe, Jacob Curtis, Vincent Bushong, and Tomas Cerny. Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761, 2022. doi:10.1109/ACCESS.2022.3160485.

38 Sheldon Smith, Ethan Robinson, Timmy Frederiksen, Trae Stevens, Tomas Cerny, Miroslav Bures, and Davide Taibi. Benchmarks for end-to-end microservices testing. *CoRR*, abs/2306.05895, 2023. doi:10.48550/ARXIV.2306.05895.

39 J. Soldani, G. Muntoni, D. Neri, and A. Brogi. The ntosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Sw. Practice and Experience*, 51(7):1591–1621, 2021. doi:10.1002/SPE.2974.

40 Jan Svacina, Vincent Bushong, Dipta Das, and Tomás Cerný. Semantic code clone detection method for distributed enterprise systems. In Maarten van Steen, Donald Ferguson, and Claus Pahl, editors, *Proceedings of the 12th International Conference on Cloud Computing and Services Science, CLOSER 2022,, Online Streaming, April 27-29, 2022*, pages 27–37. SCITEPRESS, 2022. doi:10.5220/0011032200003200.

41 Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In *Cloud Computing and Services Science*, pages 133–149, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-49432-2_7.

42 Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 153–164. INSTICC, SciTePress, 2019. doi:10.5220/0007755901530164.

43 J Thalheim, A Rodrigues, I.E. Akkus, P Bhatotia, Rm Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Middleware*, 2017. doi:10.1145/3135974.3135977.

44 A. Walker, D. Das, and T. Cerny. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21), 2020. doi:10.3390/app10217800.

45 A. Walker, I. Laird, and T. Cerny. On automatic software architecture reconstruction of microservice applications. *Information Science and Applications*, 2021.

46 Adam Wiggins. The twelve-factor app, 2017. https://12factor.net/, last accessed 1/2/2022. URL: https://12factor.net/.

47 X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Benchmarking microservice systems for software engineering research. In *40th Int. Conf.Software Engineering: Comp.*, 2018. doi:10.1145/3183440.3194991.

Modular Choreographies: Bridging Alice and Bob Notation to Java

Luís Cruz-Filipe
University of Southern Denmark, Odense, Denmark

Anne Madsen
University of Southern Denmark, Odense, Denmark

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark

Marco Peressotti
University of Southern Denmark, Odense, Denmark

Abstract

We present Modular Choreographies, a new choreographic programming language that features modular functions. Modular Choreographies is aimed at simplicity: its communication abstraction follows the simple tradition from the "Alice and Bob" notation. We develop a compiler toolchain that translates choreographies into modular Java libraries, which developers can use to participate correctly in choreographies. The key novelty is to compile through the Choral language, which was previously proposed to define object-oriented choreographies: our toolchain compiles Modular Choreographies to Choral, and then leverages the existing Choral compiler to generate Java code. Our work is the first to bridge the simplicity of traditional choreographic programming languages with the requirement of generating modular libraries in a mainstream language (Java).

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed programming languages; Applied computing → Service-oriented architectures

Keywords and phrases Choreographic Programming, Choreographies, Modularity

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.3

Supplementary Material *Software*: `https://github.com/chorlang/modular-choreographies` archived at `swh:1:dir:7550e65d0b3b2fbcf59e1e2e52eef42b79ae12c5`

Funding *Fabrizio Montesi*: Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

1 Introduction

A recognised best practice for the development of microservices [19] is to coordinate them according to *choreographies*: coordination plans that prescribe how processes in a distributed system should interact with each other, by exchanging messages [32]. However, writing programs that comply with a choreography falls under the shadow of writing correct concurrent and distributed software, which is notoriously hard even for experts [26]. This is due to the well-known state explosion problem: even for small programs, the number of possible ways in which they could interact can grow exponentially and reach unmanageable numbers [6, 35].

Choreographic programming is a programming paradigm where programs are choreographies [31]. Its aim is to relieve programmers from implementing choreographies manually, by following two steps: first, programmers can code the choreography that they wish for by using a programming language equipped with primitives that make interactions syntactically manifest; then, a compiler automatically generates a working implementation of the choreography. The theory of choreographic programming has been explored in several directions, including service-oriented computing [4], adaptability [17], cyber-physical systems [28], functional correctness [25], and security [27, 2].

Choreographic programming languages are inspired by security protocol notation (also known as “Alice and Bob” notation), which was introduced for the definition of security protocols [33]. The key primitive of these languages is the interaction term `A.expr -> B.x` which reads “`A` communicates the result of expression `expr` to `B`, which stores it in its local variable `x`”. The participants `A` and `B` are called processes, or roles [12, 3].

Until recently, implementations of choreographic programming languages mainly generated standalone systems and did not provide means to integrate the output code with mainstream development practices [31, 4, 17]. The Choral programming language was later proposed as the first choreographic programming language that can be applied to mainstream programming [20]. In Choral, a choreography is compiled to a Java library for each process described in the choreography. A developer can then import this library and invoke it to play the part of that process in a distributed system.

To achieve Java interoperability, choreographies in Choral are less abstract than usual. Developers have to take care of how communications are supported by concrete communication channels, how data types can be expressed in Java, and how choreographic functions should be structured in terms of classes and methods. Also, the simple interaction term `A.expr -> B.x` has to be written as a method invocation instead, like the following.

```
var x@B = channel.com( MyClass@A.expr() );
```

The previous line of code reads “variable `x` at `B` is assigned the result of invoking the static method `expr` of `MyClass` at `A` and passing it through an invocation of method `com` of object `channel` (which moves data from `A` to `B`)”. Understanding Choral code thus requires more knowledge. While these aspects are essential for bridging choreographies to real-world Java programs, they force designers to mix choreographies with implementation details that diminish their level of abstraction, and thus hinder reusability of choreographies for different settings.

In this paper, we bridge the gap between the traditional simplicity of choreographic languages with the practicality of Choral. Specifically, we present the following contributions:

- A new choreographic programming language, called Modular Choreographies, and its formal semantics. Modular Choreographies offers a simple choreographic syntax with the standard “Alice and Bob” communication primitive (`A.expr -> B.x`), augmented with linguistic constructs for writing parametric functions. Our design is purposefully inspired by previous choreographic languages to be familiar (more details are given in Section 2).
- A type system for Modular Choreographies, which checks that functions are invoked correctly: the processes that enact a function have access to the right data and local functions (e.g., for encryption). Our type system supports the expected property of subject reduction [37].
- An implementation of Modular Choreographies, consisting of a parser and a type checker.
- A tool that, given code in Modular Choreographies, synthesises a program in Choral. The synthesiser automatically generates classes, methods, and the necessary usages of channels in order to move data correctly as instructed by the choreography.

Taken together, our contributions and Choral enable a new development methodology for implementing software that follows choreographies correctly, which we depict below.

$$\underbrace{\boxed{\text{Modular Choreographies}} \xrightarrow{\text{Our tool}}}_{\text{This work}} \underbrace{\boxed{\text{Choral}} \xrightarrow{\text{Choral compiler}} \boxed{\text{Java Libraries}}}_{\text{From [20]}}$$

That is, developers can use Modular Choreographies to design protocols expressed in a simple choreographic language and generate valid Choral code (using our tool), from which compliant Java libraries can be automatically generated (using the compiler from [20]). This gives choreography designers the option of using a simple language, without giving up on Java interoperability. If the decisions made by our tool when synthesising the Choral code need to be refined, it can be done before the Java libraries are generated; for example, it is possible to change method names or the type used to denote data that can be transmitted (the default is `Serializable`). This is enabled by our two-step approach and, we believe, is better than editing the Java libraries directly: once we reach that level, code does not have a choreographic view anymore and therefore introducing concurrency bugs is much easier.

Using Choral as intermediate technology also gives the pragmatic advantage of reusing what already exists to a reasonable extent. In particular, we do not implement yet another procedure for compiling choreographies to separate distributed programs – a process known as Endpoint Projection [3]. This allowed us to focus on the design of Modular Choreographies and the novel aspect of connecting "Alice and Bob" choreographies to object orientation.

Structure of the Paper. Section 2 discusses relevant related work. Modular Choreographies and its type system are presented in Section 3. We recap useful background knowledge on Choral and illustrate how our implementation works in Section 4.[1] Conclusions and future work are given in Section 5.

2 Related Work

We discuss here the most related work and highlight important differences. The reader interested in an introduction to choreographic languages and their compilation can consult [32].

The design of Modular Choreographies is inspired by the theories of Procedural Choreographies [11] and Recursive Choreographies [32]. Notably, it inherits the features that processes (can) have mutable states, choreographies are parameterised over the processes that enact them, and function calls can have continuations (general recursion). Differently from these theories, Modular Choreographies includes linguistic constructs for defining and using functions that contain choreographies more modularly, motivated by the intention of using our language for practical programming. Relevant changes include: functions can have local variables; choreographies can be parameterised over the local functions that processes can run; the capability of returning values from a function, possibly at many different processes; and a type system for checking that procedures are invoked by passing arguments of the right types. We deal with the combination of mutable state, value return, and general recursion by introducing call frames to the operational semantics of choreographies. Procedures with local scopes were already present in the first choreographic programming language, Chor [4, 31], but they could not be parameterised over processes nor local functions. Other implemented choreographic programming languages that predate our work but do not offer modularity include AIOCJ [17] and hacc [9]. One choreographic programming language that does support modularity is HasChor, an embedded domain-specific language in Haskell [40]. Differently from our toolchain (and Choral, which we leverage), HasChor requires each participant to know the entire choreography in order to be executed, whereas we compile separate individual libraries; also, it injects broadcast communications for coordinating choices that

[1] At the time of this writing, the latest version of our implementation is available at `https://github.com/chorlang/modular-choreographies`.

are not always necessary in our case, while we use Choral's implementation of the "merging" operator [3] to detect that choices are communicated correctly among participants (a property known as "knowledge of choice" [5, 32]).

Our development is also based on Choral, which introduced the first integration between choreographic languages and mainstream programming abstractions [20]. In particular, Choral's types give control over the APIs that are generated in the target Java libraries. We use this control extensively to compile APIs that allow programmers to pass parameters to our Modular Choreographies (e.g., functions and data locally available at processes) and to use the values that choreographies return. Choral is also the first choreographic programming language that introduced returning values and higher-order choreographies (choreographies parameterised over choreographies), but unlike Modular Choreographies it does not offer a syntax based on the "Alice and Bob" notation and it does not come with a formal semantics.

Previous work studied theories of choreographic languages based on the "Alice and Bob" notation for several settings, including verification [8, 25, 7], cyberphysical systems [29, 28], security [27, 2], web services [3], and transactions [41]. Modular Choreographies has the potential of being an interesting basis for the future application of these theories.

Another paradigm related to choreographic programming is multitier programming, which shares the practice of compiling a program into the distributed implementations of multiple participants [43]. Differently from choreographic programming, multitier programs are not choreographies. Multitier code describes the local computation that a participant performs from its own viewpoint. However, this viewpoint can be switched to that of another participant by performing a "hop" that changes the scope of variables. For example, a function declared to be located at the client can seamlessly invoke a function declared to be located at the server. The program would then be split by a compiler to code for the client and server, and the hop would typically be implemented at runtime by a coordination middleware. The control that choreographies give over how participants communicate is known to be important for writing faithful implementations of protocols [30]. Nevertheless, at least for some multitier languages, it is possible to translate a multitier program into a choreography that implements it, which basically boils down to synthesising a precise interaction protocol that runs the multitier program [21].

There are other tools for the generation of libraries that aid developers with following communication flows, based on "global types": abstractions of choreographies that specify only the types of data to be communicated [23]. For example, in [39] global types are compiled into local specifications for the communication behaviour that each participant should implement, which are then used to generate libraries with fluid APIs that aid developers with ordering send/receive actions correctly (like `o.send(..).receive(..).send(..)`). The programmer must then implement these actions by hand. This is a less direct approach than choreographic programming, where the programmer just needs to define the source choreography and then correct implementations are automatically generated.

3 Modular Choreographies

Syntax. The syntax of Modular Choreographies is given by the grammar in Figure 1. We denote lists of similar elements with overlines e.g., writing $\overline{e}$ for $e_1, \ldots, e_n$, and optionally indicating a index variables e.g., $\overline{e_i}$. Processes are identified by process names (p, q, $\ldots$), they execute concurrently, are equipped with a local (private) memory that supports dynamic allocation, and can evaluate (local) expressions (e). We fix a minimal syntax for local expressions: v ranges over value literals, id ranges over identifiers for variables and (local)

$e ::= c \mid id(\overline{e})$	Local expressions
$c ::= v \mid id$	
$b ::= \texttt{int} \mid \texttt{string} \mid \texttt{boolean} \mid \texttt{double} \mid \ldots$	Local data types
$t ::= b \mid \overline{b}\ \texttt{->}\ b$	Local types
$C ::= \varepsilon \mid I\texttt{;}C$	Choreographies
$I ::= \textbf{var}\ p\texttt{.}id\texttt{:}b$	Variable declaration
$\mid p\texttt{.}id\ \texttt{=}\ e$	Variable assignment
$\mid \textbf{var}\ p\texttt{.}id\texttt{:}b\ \texttt{=}\ e$	
$\mid p\texttt{.}e\ \texttt{->}\ q\texttt{.}id$	Value communication
$\mid p\texttt{.}e\ \texttt{->}\ \textbf{var}\ q\texttt{.}id\texttt{:}b$	
$\mid p\ \texttt{->}\ q\texttt{[}l\texttt{]}$	Label selection
$\mid \textbf{if}\ p\texttt{.}e\ \texttt{\{}C\texttt{\}}\ \textbf{else}\ \texttt{\{}C\texttt{\}}$	Conditional
$\mid \overline{p\texttt{.(}\overline{id}\texttt{)}}\ \texttt{=}\ ID\texttt{(}\overline{p\texttt{.(}\overline{e}\texttt{)}}\texttt{)}$	Function call
$\mid \textbf{return}\ \overline{p\texttt{.(}\overline{e}\texttt{)}}$	Function return
$\mathcal{C} ::= \left\{ ID\texttt{(}\overline{p\texttt{:}\textbf{proc}\texttt{(}\overline{id\texttt{:}t}\texttt{)}}\texttt{)}\texttt{:}\texttt{(}\overline{p\texttt{:}\textbf{proc}\texttt{(}\overline{b}\texttt{)}}\texttt{)}\texttt{\{}C\texttt{\}} \right\}_{i \in I}$	Function definitions

Figure 1 Modular Choreographies, syntax (syntactic sugar is greyed).

functions, and $id(\overline{e})$ denotes application. We denote the set of value literals for a local data type Values(b). Choreographies (C) are sequences of choreographic instructions I with ε denoting an empty sequence. In the instruction $\textbf{var}\ p\texttt{.}id\texttt{:}b$, p declares a variable id with type b; and in $p\texttt{.}id\ \texttt{=}\ e$ it assigns to id of the result of evaluating the expression e; $\textbf{var}\ p\texttt{.}id\texttt{:}b\ \texttt{=}\ e$, p performs both a declaration and an assignment. In $p\texttt{.}e\ \texttt{->}\ q\texttt{.}id$, p communicates the result of evaluating the local expression e to q which stores it in its local variable id and in $p\texttt{.}e\ \texttt{->}\ \textbf{var}\ q\texttt{.}id\texttt{:}b$ the q also declares the variable id. In $p\ \texttt{->}\ q\texttt{[}l\texttt{]}$, p communicates label l (a constant distinct from values used by local computation) to q; this type of communication does not alter the state of either process and is used to propagate decisions about control flow (as common for choreographic languages). In $\textbf{if}\ p\texttt{.}e\ \texttt{\{}C\texttt{\}}\ \textbf{else}\ \texttt{\{}C\texttt{\}}$, p evaluates the (boolean) guard e and the choreography proceeds with either branch accordingly. In $\overline{p\texttt{.(}\overline{id}\texttt{)}}\ \texttt{=}\ ID\texttt{(}\overline{p\texttt{.(}\overline{e}\texttt{)}}\texttt{)}$, $\overline{p}$ call the choreographic function ID on the values obtained evaluating the formal arguments $\overline{p\texttt{.(}\overline{e}\texttt{)}}$ (each argument is at a single process) and assign the result to the variables $\overline{p\texttt{.(}\overline{id}\texttt{)}}$. Finally, in $\textbf{return}\ \overline{p\texttt{.(}\overline{e}\texttt{)}}$, each process participating in a function call returns to the caller with the values obtained by evaluating its local expressions as result. Note that choreographic functions may return multiple values at multiple processes, e.g., a function for establishing a shared secret among two parties will return a value at each of them as illustrated in the next example.

▶ **Example 1.** The Diffie-Hellman key-exchange protocol [18] allows two parties `a` and `b` to establish a shared secret `key` for symmetric encryption. The protocol assumes that the two participants share a prime number `m` and a primitive root modulo `m`, `g`, that each has a private key `privKey`, and that both can perform modular exponentiation `exp`. To implement this protocol we define a choreographic function `DH` that, for each participant, takes the arguments `privKey:int`, `g:int`, `m:int`, `exp:(int,int,int)->int` and returns a value of type `int`.

```
// Diffie-Hellman key-exchange protocol
DH(a:proc(privKey:int,g:int,m:int,exp:(int,int,int)->int),
   b:proc(privKey:int,g:int,m:int,exp:(int,int,int)->int)
):(a:proc(int),b:proc(int)) {
  a.exp(privKey,g,m) -> var b.pubKey:int; // a computes and sends its public key to b
  b.exp(privKey,g,m) -> var a.pubKey:int; // b computes and sends its public key to a
  var a.key:int = exp(privKey,pubKey,m);  // a computes its copy of the shared secret
  var b.key:int = exp(privKey,pubKey,m);  // b computes its copy of the shared secret
  return a.key, b.key;                    // a and b return the shared secret
}
```
MC Code

▶ **Example 2.** In this example we provide an implementation in MC of the Single Sign On protocol defined in [32] using Recursive Choreographies. In this protocol a client c gains access to a service s by getting its credentials checked by a third party credential authentication service a. If the check succeeds the service s will issue a token to c otherwise they will proceed with another attempt.

```
SSO(c:proc(creds:()->int),
    s:proc(newToken:()->int),
    a:proc(valid:(int)->boolean)
):(c:proc(int)) {
  c.creds() -> var a.x:int // the client sends credentials to the authority
  var c.t:int
  if a.valid(x)           // the authority checks the credentials it received
  {
     a -> s[OK]           // and informs the service it guarantees for the client
     s -> c[TOKEN]        // which informs the client it will receive a token
     s.newToken() -> c.t  // and issues a token
  } else {
     a -> s[KO]
     s -> c[ERROR]
     c.t = SSO(c.creds,s.newToken,a.valid) // retry
  }
  return c.t              // return the token at the client
}
```
MC Code

Semantics. We specify the expected behaviour of Modular Choreographies by means of an operational semantics which we will later use to validate the design of the MC type system. Our design follows the same principles used by the models we extend ([12, 11]) but dispenses from out-of-order execution of actions at distinct processes (e.g., in $p.id = e; q.id = e$, the processes p and q can, in practice, carry out their local computations independently).[2] The semantics can be readily extended to account for this aspect following the same approach based on delay actions used by *loc. cit.* and minor changes to our proofs. However, the return on this investment in complexity of the model is limited and does not result in stronger results for the aims of this work since Choral (the compilation target for MC) lacks a formal semantics.

[2] This simplification is not a first in the literature of choreographic programming: [13] defines both sequential and concurrent semantics for its choreographic language arguing that the former is precise enough to support reasoning about program correctness; [12, 14] shows that for establishing many results of interest it is sufficient to consider "head transitions" which correspond to establishing a sequential semantics.

$$\boxed{\langle \Gamma, R, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle}$$

$$\frac{p.id \notin \Gamma}{\langle \Gamma, \texttt{var}\ p.id{:}b\texttt{;}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, p.id{:}\ b, C, \Sigma[p.id := \mathrm{default}(b)] \rangle}\ \textsc{SDec}$$

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, p.id \texttt{ = } e\texttt{;}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma[p.id := v] \rangle}\ \textsc{SAss}$$

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, p.e \texttt{ -> } q.id\texttt{;}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma[q.id := v] \rangle}\ \textsc{SCom}$$

$$\frac{}{\langle \Gamma, p \texttt{ -> } q\texttt{[}l\texttt{];}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma \rangle}\ \textsc{SSel}$$

$$\frac{\Sigma(p) \vdash e \downarrow v}{\langle \Gamma, \texttt{if}\ p.e\ \texttt{\{}C_{\texttt{true}}\texttt{\}}\ \texttt{else}\ \texttt{\{}C_{\texttt{false}}\texttt{\};}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C_v \fatsemi C, \Sigma \rangle}\ \textsc{SCond}$$

$$\frac{\overline{\Sigma(p_k) \vdash e_{k,l} \downarrow c_{k,l}} \qquad ID\texttt{(}\overline{q_k\texttt{:proc(}\overline{id_{k,l}\texttt{:}t_{k,l}}\texttt{)}}\texttt{):(}\overline{q\texttt{:proc(}\overline{b}\texttt{)}}\texttt{)\{}C'\texttt{\}} \in \mathcal{C}}{\begin{array}{l}\langle \Gamma, \overline{p.\texttt{(}\overline{id}\texttt{)}} \texttt{ = } ID\texttt{(}\overline{p_k\texttt{.(}\overline{e_{k,l}}\texttt{)}}\texttt{);}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \\ \qquad \langle \Gamma, \overline{r_i\texttt{.(}\overline{id_{i,j}}\texttt{)}} \texttt{ = } \langle \overline{p_k.\overline{id_{k,l}}{:}\ t_{k,l}}, C'[\overline{p_k/q_k}], \Sigma[\overline{p_k.id_{k,l} := c_{k,l}}] \rangle\texttt{;}C, \Sigma \rangle\end{array}}\ \textsc{SCall}$$

$$\frac{\langle \Gamma', R', \Sigma' \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma' \rangle}{\langle \Gamma, \overline{p.\texttt{(}\overline{id}\texttt{)}} \texttt{ = } \langle \Gamma', R', \Sigma' \rangle\texttt{;}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, \overline{r.\texttt{(}\overline{id}\texttt{)}} \texttt{ = } \langle \Gamma', R', \Sigma' \rangle\texttt{;}C, \Sigma \rangle}\ \textsc{SCallRT}$$

$$\frac{\overline{\Sigma'(p_i) \vdash e_{i,j} \downarrow v_{i,j}}}{\langle \Gamma, \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}} \texttt{ = } \langle \Gamma', \texttt{return}\ \overline{p_i\texttt{.(}\overline{e_{i,j}}\texttt{)}}, \Sigma' \rangle\texttt{;}C, \Sigma \rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma[\overline{p_i.id_{i,j} := v_{i,j}}] \rangle}\ \textsc{SCallRet}$$

Figure 2 Modular Choreographies, semantics.

The semantics of MC is given as a transition whose states are triples $\langle \Gamma, R, \Sigma \rangle$ where:

- Γ tracks the variables and functions available at each process together with their type.
- R is a choreography term (in the syntax of MC extended with runtime terms that we will introduce below).
- Σ tracks the memory state of each process.

We represent typing environments using the grammar below assuming processes occur at most once in Γ and identifiers occur at most once in each γ.

$$\Gamma ::= \cdot \mid \Gamma, p.\gamma \qquad \gamma ::= \cdot \mid \gamma, id{:}\ t$$

We refer to Γ and γ as global and local, respectively and write $\Gamma(p)$ for the environment γ local to p in Γ (given that p occurs in Γ). We extend the syntax of MC with a runtime term for representing the call stack:

$$R ::= C \mid \overline{p\texttt{.(}\overline{id}\texttt{)}} \texttt{ = } \langle \Gamma, R, \Sigma \rangle\texttt{;}C$$

In the $\overline{p\texttt{.(}\overline{id}\texttt{)}} \texttt{ = } \langle \Gamma, R, \Sigma \rangle\texttt{;}C$, R is executed under Γ and Σ and the result of this execution will be stored under $\overline{p\texttt{.(}\overline{id}\texttt{)}}$ before continuing with C. (This design avoids the need to define frames for Γ and Σ.) The memory state Σ can be regarded as a mapping assigning each process p to its local memory state (denoted as $\Sigma(p)$). We write $\Sigma[p.id := v]$ for the state obtained by assigning v to $p.id$ in Σ. We assume that each basic type b comes with a default value and denote this value by $\mathrm{default}(b)$.

To evaluate local expressions we assume an evaluation function that takes a local state as a parameter. We write $\Sigma(p) \vdash e \downarrow c$ to denote that the local expression e evaluates to c under the state $\Sigma(p)$ (local at p).

Transitions are specified by the derivation rules reported in Figure 2. Rules SDec–SSel capture the intuition behind the different choreographic primitives given earlier. Rule SCond models the behaviour of a conditional where p chooses which choreography to execute based on the outcome of evaluating its guard (locally). In the transition target, $C_v \fatsemi C$ denotes the choreography obtained by "grafting" C onto each continuation point (ε not guarded by a return) in C_v according to the recursive definition below.

$$C \fatsemi C' = \begin{cases} C' & \text{if } C = \varepsilon \\ C & \text{if } C = \mathtt{return}\ \overline{p\texttt{.(}\overline{e}\texttt{)}}\texttt{;}\varepsilon \\ I\texttt{;}(C \fatsemi C') & \text{otherwise} \end{cases}$$

Rules SCall–SCallRet model the initialisation, execution, and termination of a function call. Rule SCall creates a fresh frame for the execution of function ID where the environment and choreography are given by the formal parameters of the function and its body where all processes (q_k) are instantiated with those provided by the call site (p_k), and the memory state is initialised with the actual parameters ($c_{k,l}$). Observe that actual parameters can be basic values (v) or identifiers (id) since functions in MC can be parametrised in the names of local functions. Rule SCallRT allows for the execution of a choreography in a call frame. Rule SCallRet models a call returning a result by removing the corresponding frame and storing the result in the state of the caller.

Typing and progress. We equip MC with a typing discipline that checks that variables, values, and functions (local or choreographic) are used according to their declared type and that results returned by choreographic functions are of the expected type. The typing judgments and derivation rules that compose the type system are summarised in Figure 3.

The typing discipline uses the local types b and t and signatures of choreographic functions found in the syntax of MC. Additionally, it uses types given by the following grammar to express whether every, some, or none of the exit points (ε and $\mathtt{return}\ \overline{p\texttt{.(}\overline{e}\texttt{)}}$) of a choreography return values of a given type.

$$S ::= \{B\} \mid \{\bot\} \mid \{\bot, B\} \qquad B ::= \overline{p\texttt{:}\mathtt{proc}\texttt{(}\overline{b}\texttt{)}}$$

A type of the first form describes a choreography where every exit point returns values according to B, a type of the second describes a choreography where every exit point is without a return instruction, and the third describes a choreography with exit point for both cases (returning values of type B or no return at all). This information will be important for choreographies with conditionals where only few branches return. To combine information about exit points in different branches of a conditional we introduce a "join" operation $S_1 \curlyvee S_2$ which is defined as $S_1 \cup S_2$ wherever S_1 and S_2 agree on the type of returned values in the sense that $B_1 \in S_1, B_2 \in S_2 \implies B_1 = B_2$. For instance, $\{\bot\} \curlyvee \{B\}$ and $\{\bot\} \curlyvee \{\bot, B\}$ both yield $\{\bot, B\}$ whereas $\{p.\mathtt{int}\} \curlyvee \{p.\mathtt{boolean}\}$ is undefined.

Additionally to global (Γ) and local (γ) typing environments, the typing discipline relies on separate typing environments (Δ) for recording the choreographic functions available and their signature. These environments are represented using the grammar below under the assumption that each ID occurs at most once.

$$\Delta ::= \cdot \mid \Delta, ID\colon \overline{q_i\texttt{:}\mathtt{proc}\texttt{(}\overline{t_{i,k}}\texttt{)}} \to \overline{q_i\texttt{:}\mathtt{proc}\texttt{(}\overline{b_{i,j}}\texttt{)}}$$

$\boxed{\gamma \vdash e\colon t}$

$$\frac{v \in \text{Values}(b)}{\gamma \vdash v\colon b}\,\textsc{TLit} \qquad \frac{id\colon \overline{b}\ \texttt{->}\ b \in \gamma}{\gamma \vdash id\colon \overline{b}\ \texttt{->}\ b}\,\textsc{TId} \qquad \frac{\gamma \vdash id\colon b_1,\dots,b_n\ \texttt{->}\ b \quad \overline{\gamma \vdash e_i\colon b_i}}{\gamma \vdash id(e_1,\dots,e_n)\colon b}\,\textsc{TApp}$$

$\boxed{\Gamma \vdash \Sigma}$

$$\frac{}{\cdot \vdash \Sigma} \qquad \frac{\Gamma(p) \vdash \Sigma(p)(id)\colon t \quad \Gamma \vdash \Sigma}{\Gamma, p.id\colon t \vdash \Sigma}$$

$\boxed{\Delta;\Gamma \vdash R\colon S}$

$$\frac{}{\Delta;\Gamma \vdash \varepsilon\colon \{\bot\}}\,\textsc{TNil} \qquad \frac{\overline{\Gamma(p_i) \vdash e_{i,j}\colon b_{i,j}}}{\Delta;\Gamma \vdash \textbf{return}\ \overline{p_i.(\overline{e_{i,j}})}\texttt{;}\colon \{\overline{p_i\texttt{:}\textbf{proc}(\overline{b_{i,j}})}\}}\,\textsc{TRet}$$

$$\frac{id \notin \gamma \quad \Delta;\Gamma, p.(\gamma, id\colon b) \vdash C\colon S}{\Delta;\Gamma, p.\gamma \vdash \textbf{var}\ p.id\texttt{:}b\texttt{;}C\colon S}\,\textsc{TDec} \qquad \frac{\Gamma(p) \vdash e\colon b \quad \Gamma(q) \vdash id\colon b \quad \Delta;\Gamma \vdash C\colon S}{\Delta;\Gamma \vdash p.e\ \texttt{->}\ q.id\texttt{;}C\colon S}\,\textsc{TCom}$$

$$\frac{\Gamma(p) \vdash id\colon b \quad \Gamma(p) \vdash e\colon b \quad \Delta;\Gamma\colon b \vdash C\colon S}{\Delta;\Gamma \vdash p.id\ \texttt{=}\ e\texttt{;}C\colon S}\,\textsc{TAss} \qquad \frac{p, q \in \Gamma \quad \Delta;\Gamma \vdash C\colon S}{\Delta;\Gamma \vdash p\ \texttt{->}\ q[l]\texttt{;}C\colon S}\,\textsc{TSel}$$

$$\frac{\Gamma(p) \vdash e\colon \texttt{boolean} \quad \Delta;\Gamma \vdash C_i\colon S_i \quad \bot \notin (S_1 \curlyvee S_2) \implies C_3 = \varepsilon}{\Delta;\Gamma \vdash \textbf{if}\ p.e\ \{C_1\}\ \textbf{else}\ \{C_2\}\texttt{;}C_3\colon S_1 \curlyvee S_2 \curlyvee S_3}\,\textsc{TCond}$$

$$\frac{\overline{\Gamma \vdash p_k.e_{i,k}\colon t_{i,k}} \quad \overline{\Gamma \vdash p_i.id_{i,j}\colon b_{i,j}} \quad \Delta;\Gamma \vdash C\colon S}{\Delta, ID\colon \overline{q_i\texttt{:}\textbf{proc}(\overline{t_{i,k}})} \to \overline{q_i\texttt{:}\textbf{proc}(\overline{b_{i,j}})};\Gamma \vdash \overline{p_i.(\overline{id_{i,j}})}\ \texttt{=}\ ID(\overline{p_i.(\overline{e_{i,k}})})\texttt{;}C\colon S}\,\textsc{TCall}$$

$$\frac{\Delta;\Gamma' \vdash R\colon \{\overline{p_i\texttt{:}\textbf{proc}(\overline{b_{i,j}})}\} \quad \Gamma' \vdash \Sigma' \quad \overline{\Gamma \vdash p_i.id_{i,j}\colon b_{i,j}} \quad \Delta;\Gamma \vdash C\colon S}{\Delta;\Gamma \vdash \overline{p_i.(\overline{id_{i,j}})}\ \texttt{=}\ \langle \Gamma', R, \Sigma\rangle\texttt{;}C\colon S}\,\textsc{TCallRT}$$

$\boxed{\Delta \vdash \mathcal{C}}$

$$\frac{ID\colon \overline{p_i\texttt{:}\textbf{proc}(\overline{t_{i,j}})} \to \overline{p_i\texttt{:}\textbf{proc}(\overline{b_{i,k}})} \in \Delta \quad \Delta; \overline{p_i.id_{i,j}\colon t_{i,j}} \vdash C\colon \{\overline{p_i\texttt{:}\textbf{proc}(\overline{b_{i,k}})}\}}{\Delta \vdash ID(\overline{p_i\texttt{:}\textbf{proc}(\overline{id_{i,j}\texttt{:}t_{i,j}})})\texttt{:}\overline{p_i\texttt{:}\textbf{proc}(\overline{b_{i,k}})}\ \{C\}}\,\textsc{TDef}$$

Figure 3 Modular Choreographies, typing.

Typing judgments are of four forms. A judgment $\gamma \vdash e\colon t$ indicates that the local expression e has type t under the local typing environment γ. A judgment $\Gamma \vdash \Sigma$ signifies that the memory state Σ is consistent with the declarations in Γ i.e., that maps each symbol defined in Γ to an element of the expected type. A judgment $\Delta;\Gamma \vdash R\colon S$ indicates that under Δ and Γ, the (runtime) choreography R has return type S. Rules TNil and TRet type the exit points of a choreography and Rule TCond combines the information about exit points in each branch (C_1, C_2) and in the continuation (C_3) ensuring that if values are returned they are all of the same type (by definition of $\curlyvee$) and that the continuation is empty whenever all exit points in both branches return (i.e., $\bot \notin C_1 \curlyvee C_2$). Rule TCall ensures that the called function is declared and that the actual parameters ($p_i.e_{i,k}$) and target variables ($p_i.id_{i,j}$) are of the types specified by the function signature ($q_i.t_{i,k}$ for formal parameters and $q_i.b_{i,k}$ for results, respectively). Rule TCall ensures that the type of the target variables ($p_i.id_{i,j}$) for the call under execution (R) coincides with the type of the values returned by (every) exit point for the call. The remaining inference rules for

$\Delta;\Gamma \vdash R\colon S$ follow the intuition behind the different choreographic primitives. Observe that rule selection is completely syntax-driven: the outermost construct of a term identifies uniquely identifies which rule to apply. Finally, a judgment $\Delta \vdash \mathcal{C}$ means that the function definitions in $\mathcal{C}$ agree with the signatures declared in Δ and are well-typed. The last property is captured by Rule TDEF which states that the body of a choreographic function has the expected return type under Δ and the environment Γ given by its formal arguments.

In the remainder we assume subject reduction for local expressions i.e., that evaluating a well-typed expression in a memory state compatible with the environment used to type the expression yields a result of the expected type.

▶ **Assumption 3.** *If $\Gamma \vdash \Sigma$, $\Gamma(p) \vdash e\colon t$, and, $\Sigma(p) \vdash e \downarrow c$, then $\Gamma(p) \vdash c\colon t$.*

Under Assumption 3, well-typed configurations can only evolve into well-typed configurations i.e., MC enjoys subject reduction.

▶ **Theorem 4** (Subject reduction). *If $\Delta \vdash \mathcal{C}$, $\Delta;\Gamma \vdash R\colon S$, $\Gamma \vdash \Sigma$ and $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$, then $\Gamma' \vdash \Sigma'$ and $\Delta;\Gamma' \vdash R'\colon S'$ for $S' \subseteq S$.*

Proof. By nested induction on the derivation of $\Delta;\Gamma \vdash R\colon S$ and $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$.

- Consider the case of Rule TDEC. Then, $R = \texttt{var}\ p\texttt{.}id\texttt{:}b\texttt{;}C$, $p\texttt{.}id \notin \Gamma$, and $\Delta;\Gamma \vdash C\colon S$. The only case for $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$ is that of Rule SDEC and thus $\Gamma' = \Gamma, p\texttt{.}id\colon b$, $R' = C$, and $\Sigma' = \Sigma[p\texttt{.}id := \text{default}(b)]$. By definition $\text{default}(b) \in \text{Values}(b)$ and thus $\Gamma' \vdash \Sigma'$.
- Consider the case of Rule TCOM. Then, $R = p\texttt{.}e\ \texttt{->}\ q\texttt{.}id\texttt{;}C$, $\Gamma(p) \vdash e\colon b$, $\Gamma(q) \vdash id\colon b$, and $\Delta;\Gamma \vdash C\colon S$. The only case for $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$ is that of Rule SCOM and thus $\Gamma' = \Gamma$, $R' = C$, and $\Sigma' = \Sigma[q\texttt{.}id := v]$ where $\Sigma(p) \vdash e \downarrow v$. It follows from $\Gamma(p) \vdash e\colon b$ and Assumption 3 that $\Gamma' \vdash \Sigma'$.
- Consider the case of Rule TCOND. Then, $R = \texttt{if}\ p\texttt{.}e\ \texttt{\{}C_1\texttt{\}}\ \texttt{else}\ \texttt{\{}C_2\texttt{\};}C_3$, $\Gamma(p) \vdash e\colon \texttt{boolean}$, and $\Delta;\Gamma \vdash C_i\colon S_i$ for $i \in \{1,2,3\}$. It follows from $\Gamma(p) \vdash e\colon \texttt{boolean}$, $\Gamma \vdash \Sigma$ and Assumption 3 that $\Sigma(p) \vdash e \downarrow v$ for $v \in \text{Values}(\texttt{boolean})$ and thus that the only case for $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$ is that of Rule SCOND. If $\Sigma(p) \vdash e \downarrow \texttt{true}$ then $R' = C_1 \mathbin{⨟} C_3$, $S' = S_1 \curlyvee S_3$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma$. To derive $\Delta;\Gamma' \vdash C'\colon S'$ it suffices to take all applications Rule TNIL in the derivation of $\Delta;\Gamma \vdash C_1\colon S_1$ that correspond to an occurrence of ε replaced with C_3 in $C_1 \mathbin{⨟} C_3$ and replace them with a copy of the derivation for $\Delta;\Gamma \vdash C_3\colon S_3$. By definition of $\curlyvee$, $S' \subseteq S$. The case for $\Sigma(p) \vdash e \downarrow \texttt{false}$ is similar.
- Consider the case of Rule TCALLRT. Then, $R = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}} \texttt{=} \langle \Gamma_1, R_1, \Sigma_1\rangle\texttt{;}C$, $\Delta;\Gamma_1 \vdash R_1\colon \{\overline{p_i\texttt{:proc(}\overline{b_{i,j}}\texttt{)}}\}$, $\Gamma_1 \vdash \Sigma_1$, $\overline{\Gamma \vdash p_i\texttt{.}id_{i,j}\colon b_{i,j}}$, and $\Delta;\Gamma \vdash C\colon S$. Any derivation of $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$ starts with an application of either Rule SCALLRET or Rule SCALLRT.
  - In the first case, $R_1 = \texttt{return}\ \overline{p_i\texttt{.(}\overline{e_{i,j}}\texttt{)}}$, $R' = C$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma[\overline{p_i.id_{i,j} := v_{i,j}}]$ where $\overline{\Sigma_1(p_i) \vdash e_{i,j} \downarrow v_{i,j}}$, and $S' = S$. By $\Delta;\Gamma_1 \vdash R_1\colon \{\overline{p_i\texttt{:proc(}\overline{b_{i,j}}\texttt{)}}\}$ and Assumption 3, $v_{i,j} \in \text{Values}(b_{i,j})$ and thus $\Gamma' \vdash \Sigma'$.
  - In the second case, $\langle \Gamma_1, R_1, \Sigma_1\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma_2, R_2, \Sigma_2\rangle$, $R' = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}} \texttt{=} \langle \Gamma_2, R_2, \Sigma_2\rangle\texttt{;}C$, $\Gamma' = \Gamma$, $\Sigma' = \Sigma$, and $S' = S$. By inductive hypothesis, $\Gamma_2 \vdash \Sigma_2$ and $\Delta;\Gamma_2 \vdash R_2\colon \{\overline{p_i\texttt{:proc(}\overline{b_{i,j}}\texttt{)}}\}$ and by Rule TCALLRT, $\Delta;\Gamma' \vdash R'\colon S'$.
- Remaining cases are straightforward adaptations of the ones above. ◀

Well-typed choreographies enjoy progress: a choreography equipped with memory compatible with its typing environment can always perform a transition unless it is terminated (it consists of a return instruction or no instruction at all).

▶ **Theorem 5** (Progress). *If $\Delta \vdash \mathcal{C}$, $\Delta;\Gamma \vdash R\colon S$, $\Gamma \vdash \Sigma$, then either*
- $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', R', \Sigma'\rangle$ *or*
- *R is either ε or a return (i.e., has form* $\texttt{return}\ \overline{p\texttt{.}(\overline{e})\texttt{;}}$*).*

Proof. By induction on the derivation for $\Delta;\Gamma \vdash R\colon S$.
- Consider the case of Rule TNil. Then, $R = \varepsilon$ and $\langle \Gamma, \varepsilon, \Sigma\rangle$ does not admit any transition.
- Consider the case of Rule TRet. Then, $R = \texttt{return}\ \overline{p\texttt{.}(\overline{e})\texttt{;}}$ and $\langle \Gamma, R, \Sigma\rangle$ does not admit any transition.
- Consider the case of Rule TDec. Then, $R = \texttt{var}\ p\texttt{.}id\texttt{:}b\texttt{;}C$, $p\texttt{.}id \notin \Gamma$, and $\Delta;\Gamma \vdash C\colon S$. By Rule SDec, $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma', C, \Sigma'\rangle$ where $\Gamma' = \Gamma, p\texttt{.}id\colon b$ and $\Sigma' = \Sigma[p\texttt{.}id := \mathrm{default}(b)]$.
- Consider the case of Rule TCom. Then, $R = p\texttt{.}e\ \texttt{->}\ q\texttt{.}id\texttt{;}C$, $\Gamma(p) \vdash e\colon b$, $\Gamma(q) \vdash id\colon b$, and $\Delta;\Gamma \vdash C\colon S$. It follows from $\Gamma(p) \vdash e\colon b$, $\Gamma \vdash \Sigma$ and Assumption 3 that $\Sigma(p) \vdash e \downarrow v$ for $v \in \mathrm{Values}(b)$, By Rule SCom, $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma'\rangle$ where $\Sigma' = \Sigma[q\texttt{.}id := v]$.
- Consider the case of Rule TCond. Then, $R = \texttt{if}\ p\texttt{.}e\ \{C_1\}\ \texttt{else}\ \{C_2\}\texttt{;}C_3$, $\Gamma(p) \vdash e\colon \texttt{boolean}$, and $\Delta;\Gamma \vdash C_i\colon S_i$ for $i \in \{1,2,3\}$. It follows from $\Gamma(p) \vdash e\colon \texttt{boolean}$ and $\Gamma \vdash \Sigma$ that $\Sigma(p) \vdash e \downarrow v$ for $v \in \mathrm{Values}(\texttt{boolean})$. By Rule SCond, $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, R', \Sigma'\rangle$ and where R' is $C_1 \fatsemi C_3$ if $\Sigma(p) \vdash e \downarrow \texttt{true}$ and $C_2 \fatsemi C_3$ otherwise.
- Consider the case of Rule TCall. Then, $R = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}}\texttt{ = }ID(\overline{p_i\texttt{.(}\overline{e_{i,k}}\texttt{)}})\texttt{;}C$, $\Delta;\Gamma \vdash C\colon S$, $\Delta = \Delta', ID\colon \overline{q_i\texttt{:proc(}\overline{t_{i,k}}\texttt{)}} \to \overline{q_i\texttt{:proc(}\overline{b_{i,j}}\texttt{)}}$, $\overline{\Gamma(p_i) \vdash e_{i,k}\colon t_{i,k}}$, and $\overline{\Gamma(p_i) \vdash id_{i,j}\colon b_{i,j}}$. It follows that $ID(\overline{q_i\texttt{:proc(}\overline{id_{i,k}\texttt{:}t_{i,k}}\texttt{)}})\texttt{:(}\overline{q_i\texttt{:proc(}\overline{id_{i,j}\texttt{:}b_{i,j}}\texttt{)}}\texttt{)\{}C'\texttt{\}} \in \mathcal{C}$, $\overline{\Sigma(p_i) \vdash e_{i,k} \downarrow c_{i,k}}$, and $\overline{\Gamma(p_i) \vdash c_{i,k}\colon t_{i,k}}$. By Rule SCall, there is $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, R', \Sigma\rangle$ with $R' = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}}\texttt{ = }\langle \overline{p_i\texttt{.}id_{i,k}\colon t_{i,k}}, C'[\overline{p_i/q_i}], \Sigma[\overline{p_i.id_{i,k} := c_{i,k}}]\rangle\texttt{;}C$.
- Consider the case of Rule TCallRT. Then, $R = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}}\texttt{ = }\langle \Gamma_1, R_1, \Sigma_1\rangle\texttt{;}C$, $\Delta;\Gamma_1 \vdash R_1\colon \{\overline{p_i\texttt{:proc(}\overline{b_{i,j}}\texttt{)}}\}$, $\Gamma_1 \vdash \Sigma_1$, $\overline{\Gamma \vdash p_i\texttt{.}id_{i,j}\colon b_{i,j}}$, and $\Delta;\Gamma \vdash C\colon S$. Because of its type, R_1 cannot be ε. By inductive hypothesis, either $\langle \Gamma_1, R_1, \Sigma_1\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma_2, R_2, \Sigma_2\rangle$ or $R_1 = \texttt{return}\ \overline{p_i\texttt{.(}\overline{e_{i,j}}\texttt{);}}$. Assume the first case. By Rule SCallRT, $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, R', \Sigma\rangle$ for $R' = \overline{p_i\texttt{.(}\overline{id_{i,j}}\texttt{)}}\texttt{ = }\langle \Gamma_2, R_2, \Sigma_2\rangle\texttt{;}C$. Assume the second case. By hypothesis, $\overline{\Sigma_1(p_i) \vdash e_{i,j} \downarrow v_{i,j}}$ and by Rule SCallRet, $\langle \Gamma, R, \Sigma\rangle \longrightarrow_{\mathcal{C}} \langle \Gamma, C, \Sigma[\overline{p_i.id_{i,j} := v_{i,j}}]\rangle$.
- Remaining cases are straightforward adaptations of the ones above. ◀

4 Translation to Choral

In this section we describe how our tool can translate choreographies in our language to Java implementations via the Choral language [20].

Choral is an object-oriented choreographic programming language: in Choral classes and interfaces represent distributed data types parametric in the processes participating in them. For instance, the snippet below contains the definition of a Choral class `Tuple2` that implements a generic tuple distributed over two processes (`A` and `B`) each holding one of the two components of the tuple (`left` is stored at `A` and `right` at `B`).

```
public class Tuple2@(A,B)<L@C,R@D> {
  public final L@A left;  public final R@B right;
  public Tuple2(L@A left, R@B right) {
    this.left = left; this.right = right;
  }
}
```
Choral Code

The Choral compiler generate Java implementation for each participant of a Choral type. For instance, `Tuple2` is compiled to the following Java classes.

```java
// Implementation of Tuple2 for A
public class Tuple2_A<L,R> {
  public final L left;
  public Tuple2_A(L left) {
    this.left = left;
  }
}
```
Java Code

```java
// Implementation of Tuple2 for B
public class Tuple2_B<L,R> {
  public final R right;
  public Tuple2_B(R right) {
    this.right = right;
  }
}
```
Java Code

Types like `Tuple2` above allow us to represent in Choral the distributed return of MC. For instance, `return (a.true,b.5)` is translated into `return new Tuple2@(A,B)<Boolean, Integer>(true@A,5@B)`.

Differently from other choreographic languages like MC, Choral does not fix a communication primitive. Instead, communication can be programmed directly: The Choral standard library provides a framework with several kinds of channels but programmers can provide their own by writing them directly in Choral or by wrapping implementations written in Java into Choral types. For instance, the Choral standard library defines the following interface to represent a generic channel between two processes, abstracted by `A` and `B`, for transmitting data of a given type, abstracted by the type parameter `T`.

```
public interface SymDataChannel@(A,B)<T@C> {
  public <M@C extends T@C> M@B com(M@A message);
  public <M@C extends T@C> M@A com(M@B message);
}
```
Choral Code

Data transmission is performed by invoking the (overloaded) method `com` which takes any value of a subtype `M` of `T` located at one process, say `A`, and returns a value of the same type at the other process, say `B`. In the example below, `A` transmits `5` to `B` using the channel `chAB`.

```
SymDataChannel@(A,B)<Serializable> chAB = /* ... */
Integer@B x = chAB.<Integer>com(5@A);
```
Choral Code

The interface `SymChannel` extends `SymDataChannel` with methods for selecting labels (which are represented as enums).

```
public interface SymChannel@(A,B)<T@C> extends SymDataChannel@(A,B)<T> {
  public <L@C extends Enum@C<L>> L@B select(L@A label);
  public <L@C extends Enum@C<L>> L@A select(L@B label);
}
```
Choral Code

Building on these features of Choral (and standard constructs like assignments and conditionals) we can translate any choreography function written in MC into Choral. Given a choreography function, our translation generates a Choral class with the same name and parametrised in the participants of the function. For instance, the choreographic function `DH` from Example 1 is compiled to a Choral class `DH@(A,B)`, as shown in the example below. This class exposes a single static method `run` which implements the behaviour of the choreographic function. In addition to the parameters specified by the choreography function, `run` takes a `SymChannel` for each pair of participants (we follow a lexicographic order to avoid ambiguity). The body of the method is generated by mapping communications and selections to invocations of the methods exposed by these channels, and other constructs homomorphically. We illustrate the translation in the examples below, we include the original MC line as comments for readability.

▶ **Example 6.** Consider the choreographic function `DH` from Example 1, our tool generates the following Choral implementation for it (comments are added for readability).

```
public class DH@(A,B) {
  public static Tuple2@(A,B)<Integer,Integer> run(
    /* channel between A and B */
    SymChannel@(A,B)<Serializable> chAB,
    /* parameters for A: proc(privKey:int,g:int,m:int,exp:(int,int,int)->int) */
    Integer@A privKey_A, Integer@A g_A, Integer@A m_A,
    Function3@A<Integer,Integer,Integer,Integer> exp_A,
    /* parameters for B: proc(privKey:int,g:int,m:int,exp:(int,int,int)->int) */
    Integer@B privKey_B, Integer@B g_B, Integer@B m_B,
    Function3@B<Integer,Integer,Integer,Integer> exp_B
  ) {
    // a computes and sends its public key to b
    // a.exp(privKey, g, m) -> var b.pubKey:int;
    Integer@B pubKey_B = chAB.<Integer>com( exp_A(privKey_A, g_A, m_A) );
    // b computes and sends its public key to a
    // b.exp(privKey, g, m) -> var a.pubKey:int;
    Integer@A pubKey_A = chAB.<Integer>com( exp_B(privKey_B, g_B, m_B) );
    // a computes its copy of the shared secret
    // var a.key:int = exp(privKey, pubKey, m);
    Integer@A key_A = exp_A(privKey_A,pubKey_A,m_A) );
    // b computes its copy of the shared secret
    // var b.key:int = exp(privKey, pubKey, m);
    Integer@B key_B = exp_B(privKey_B, pubKey_B, m_B) );
    // a and b return the shared secret
    // return a.key, b.key;
    return new Tuple2@(A,B)<Integer,Integer>(key_A,key_B);
  }
}
```

Choral Code

▶ **Example 7.** Consider the choreographic function `SSO` from Example 2, our tool generates the following implementation in Choral for it.

```
public class SSO@(C, S, A) {
  public static Integer@C run(
    /* channels */
    SymChannel@(A, C)<Serializable> chAC,
    SymChannel@(C, S)<Serializable> chCS,
    /* parameters for C: proc(creds:()->int) */
    Supplier@C<Integer> creds_C,
    /* parameters for S: proc(newToken:()->int) */
    Supplier@S<Integer> newToken_S,
    /* parameters for A: proc(valid:(int)->boolean) */
    Function@A<Integer,Boolean> valid_A,
  ) {
    // c.creds() -> var a.x:int
    Integer@A x = chAC.<Integer>com( creds_C() );
    // var c.t:int;
    Integer@C t_C;
    // if a.valid(x)
    if (valid_A(x)) {
      // a -> s[OK];
      chAS.<Label>select(Label@A.OK);
      // s -> c[TOKEN];
      chCS.<Label>select(Label@S.TOKEN);
      // s.newToken() -> c.t;
      t_C = chCS.<Integer>com( newToken_S() );
    } else {
      // a -> s[KO];
      chAS.<Label>select(Label@A.KO);
      // s -> c[ERROR];
```

Choral Code

```
      chCS.<Label>select(Label@S.ERROR);
      // c.t = SSO(c.creds, s.newToken, a.valid);
      t_C = SSO@(C, S, A).run(chAC, chCS, creds_C, newToken_S, valid_A);
    }
    // return c.t;
    return t_C;
  }
}
```

Choral Code

The Choral code generated by our translation is correct and well-typed, provided the original choreography is also well-typed. To enforce this requirement, our tool implements the typing discipline described in Section 3. First, the typing environment Δ is populated using the signatures of the function definitions provided to the tool. Then, each procedure definition if checked as specified by Rule TDEF using the syntax to guide the selection of the corresponding typing rule.

To obtain Java implementations, we then rely on the Choral compiler.

▶ **Example 8.** Continuing Example 6, the Choral compiler produces the following implementation for `A` (the one for `B` is similar).

```
public class DH_A {
  public static Tuple2_A<Integer,Integer> run(
    SymChannel_A<Serializable> chAB,
    Integer privKey_A, Integer g_A, Integer m_A,
    Function3<Integer,Integer,Integer,Integer> exp_A
  ) {
    chAB.com<Integer>( exp_A(privKey_A,g_A,m_A) );
    Integer pubKey_A = chAB.com<Integer>();
    Integer key_A = exp_A(privKey_A,pubKey_A,m_A) );
    return new Tuple2_A<Integer,Integer>(key_A);
  }
}
```

Java Code

▶ **Example 9.** Continuing Example 7, the Choral compiler produces the following implementations for `A`, `C`, and `S`.

```
public class SSO_A {
  public static void run(
    SymChannel_A<Serializable> chAC,
    SymChannel_A<Serializable> chAS,
    Function<Integer,Boolean> valid_A
  ) {
    Integer x = chAC.<Integer>com();
    if ( valid_A(x) ) {
      chAS.<Label>select(Label.OK);
    } else {
      chAS.<Label>select(Label.KO);
      SSO.run(chAC, chAS, valid_A);
    }
    return;
  }
}
```

Java Code

```
public class SSO_C {
  public static Integer run(
    SymChannel_B<Serializable> chAC,
    SymChannel_A<Serializable> chCS,
    Suplier<Integer> creds_C
  ) {
    chAC.<Integer>com( creds_C() );
    Integer t_C;
    switch ( chCS.<Label>select() ):
      case Label.TOKEN -> {
        t_C = chCS.<Integer>com();
      }
      case Label.ERROR -> {
        t_C = SSO.run(chAC, chCS,
             creds_C);
      }
    }
    return t_C;
  }
}
```
Java Code

```
public class SSO_S {
  public static void run(
    SymChannel_B<Serializable> chAS,
    SymChannel_B<Serializable> chCS,
    Suplier<Integer> newToken_S
  ) {
    switch ( chAS.<Label>select() ):
      case Label.OK -> {
        chCS.<Label>select(Label.TOKEN);
        chCS.<Integer>com( newToken_S()
            );
      }
      case Label.KO -> {
        chCS.<Label>select(Label.ERROR);
        SSO.run(chAS, chCS, newToken_S);
      }
    }
    return;
  }
}
```
Java Code

5 Conclusion and Future Work

Modular Choreographies brings the simplicity of choreographic programming based on the "Alice and Bob" communication abstraction one step nearer to practical programming.

An immediate direction for future work will be extending our language with more features, trying to keep the syntax as simple as possible. Inspiration for this could naturally come from theoretical models of choreographic languages, as those found in the research lines of choreographic programming and multiparty session types (abstract choreographies without computation) [32, 23, 24, 1]. For example, adding features for dynamic topologies (e.g., spawning new processes) is important for capturing some parallel algorithms [10, 34, 42], nondeterministism is crucial to capturing patterns like barriers and producers/consumers [32], and mixed choices or unordered communication sets are relevant for modelling exchanges [32, 36, 8, 13]. Another source of inspiration might come from the developments of choreographic programming languages like Choral, whose expressivity benefits significantly from the integration with mainstream programming abstractions like functions and objects (and their related type theories). For example, Choral is expressive enough to implement full-duplex asynchronous communications, where interactions can be triggered by any participant and be interleaved freely [30]. Another interesting line of future work regards formalising the guarantees provided by Modular Choreographies. Choreographic programming languages equipped with general recursion, like ours, are known to provide deadlock-freedom [32]. To prove this with confidence, one could extend previous formalisations of choreographic programming in theorem provers, as those given in [15, 14, 16, 38, 22].

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

2 Alessandro Bruni, Marco Carbone, Rosario Giustolisi, Sebastian Mödersheim, and Carsten Schürmann. Security protocols as choreographies. In Daniel Dougherty, José Meseguer, Sebastian Alexander Mödersheim, and Paul D. Rowe, editors, *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday*, volume 13066 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2021. doi:10.1007/978-3-030-91631-2_5.

3 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012. doi:10.1145/2220365.2220367.

4 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

5 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:24)2012.

6 Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.

7 Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. Reasoning about choreographic programs. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13908 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2023. doi:10.1007/978-3-031-35361-1_8.

8 Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In Javier Esparza and Andrzej S. Murawski, editors, *FoSSaCS*, volume 10203 of *LNCS*, pages 424–440. Springer, 2017. doi:10.1007/978-3-662-54458-7_25.

9 Luís Cruz-Filipe, Lovro Lugovic, and Fabrizio Montesi. Certified compilation of choreographies with hacc. In Marieke Huisman and António Ravara, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, volume 13910 of *Lecture Notes in Computer Science*, pages 29–36. Springer, 2023. doi:10.1007/978-3-031-35355-0_3.

10 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In *FORTE*, volume 9688 of *LNCS*, pages 114–123. Springer, 2016. doi:10.1007/978-3-319-39570-8_8.

11 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017. doi:10.1007/978-3-319-60225-7_7.

12 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/J.TCS.2019.07.005.

13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Communications in choreographies, revisited. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1248–1255. ACM, 2018. doi:10.1145/3167132.3167267.

14 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan,*

September 8-10, 2021, Proceedings, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

15 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ITP.2021.15.

16 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.

17 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.

18 Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.

19 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, 2017. doi:10.1007/978-3-319-67425-4_12.

20 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.*, Nov 2023. Accepted. doi:10.1145/3632398.

21 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ECOOP.2021.22.

22 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi:10.1145/3498684.

23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1):9, 2016. doi:10.1145/2827695.

24 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

25 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies—computing preconditions in choreographic programming. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Proceedings*, volume To appear of *Lecture Notes in Computer Science*. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.

26 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, pages 517–530. ACM, 2016. doi:10.1145/2872362.2872374.

27 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

28 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

29 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016. doi:10.1007/978-3-319-39570-8_13.

30 Lovro Lugovic and Fabrizio Montesi. Real-world choreographic programming: An experience report. *CoRR*, abs/2303.03983, 2023. doi:10.48550/ARXIV.2303.03983.

31 Fabrizio Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013. URL: https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

32 Fabrizio Montesi. *Introduction to Choreographies.* Cambridge University Press, 2023.

33 R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, dec 1978. doi:10.1145/359657.359659.

34 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised scribble. *Serv. Oriented Comput. Appl.*, 9(3-4):269–284, 2015. doi:10.1007/S11761-014-0172-8.

35 Peter W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018. doi:10.1007/978-3-319-99725-4_5.

36 Kirstin Peters and Nobuko Yoshida. On the expressiveness of mixed choice sessions. In Valentina Castiglioni and Claudio Antares Mezzina, editors, *Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, EXPRESS/SOS 2022, and 19th Workshop on Structural Operational Semantics Warsaw, Poland, 12th September 2022*, volume 368 of *EPTCS*, pages 113–130, 2022. doi:10.4204/EPTCS.368.7.

37 Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, MA, USA, 2002.

38 Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITP.2022.27.

39 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ECOOP.2017.24.

40 Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming for all (functional pearl). *CoRR*, abs/2303.00924, 2023. doi:10.48550/ARXIV.2303.00924.

41 Ton Smeele and Sung-Shik Jongmans. Choreographic programming of isolated transactions. *Electronic Proceedings in Theoretical Computer Science*, 378:49–60, apr 2023. doi:10.4204/EPTCS.378.5.

42 Vasco T. Vasconcelos, Francisco Martins, Hugo-Andrés López, and Nobuko Yoshida. A type discipline for message passing parallel programs. *ACM Trans. Program. Lang. Syst.*, 44(4):26:1–26:55, 2022. doi:10.1145/3552519.

43 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Comput. Surv.*, 53(4):81:1–81:35, 2020. doi:10.1145/3397495.

ZVAX – A Microservice Reference Architecture for Nation-Scale Pandemic Management

Oliver Cvetkovski
Zurich University of Applied Sciences, Winterthur, Switzerland
University of St. Cyril and Methodius, Skopje, North Macedonia

Carlo Field
Zurich University of Applied Sciences, Winterthur, Switzerland

Davide Trinchi
Zurich University of Applied Sciences, Winterthur, Switzerland

Christof Marti
Zurich University of Applied Sciences, Winterthur, Switzerland

Josef Spillner[1]
Zurich University of Applied Sciences, Winterthur, Switzerland

Abstract

Domain-specific Microservice Reference Architectures (MSRA) have become relevant study objects in software technology. They facilitate the technical evaluation of service designs, compositions patterns and deployment configurations in realistic operational practice. Current knowledge about MSRA is predominantly confined to business domains with modest numbers of users per application. Due to the ongoing massive digital transformation of society, people-related online services in e-government, e-health and similar domains must be designed to be highly scalable at entire nation level at affordable infrastructure cost. With ZVAX, we present such a service in the e-health domain. Specifically, the ZVAX implementation adheres to an MSRA for pandemic-related processes such as vaccination registration and passenger locator form submission, with emphasis on selectable levels of privacy. We argue that ZVAX is valuable as study object for the training of software engineers and for the debate on arbitrary government-to-people services at scale.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Computer systems organization → Availability; Applied computing → E-government

Keywords and phrases Nation-scale applications, microservices, digital transformation

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.4

Related Version *Previous Version*: https://www.conf-micro.services/2022/papers/paper_6.pdf

Supplementary Material *Software*: https://doi.org/10.5281/zenodo.7869751

1 Introduction

Microservice-based software applications are expected to have many advantages. They are supposed to be easier to develop with distributed teams of software engineers using polyglot implementations, to allow for more customisation through flexible service composition, and to be more aligned with business-critical runtime properties such as high scalability and resilience.

[1] Corresponding author

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 4; pp. 4:1–4:16
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the first years of research on microservices, practical applications to prove architecture-related hypotheses and to implement new and innovative concepts were lacking. A first applications overview was assembled in 2017 [7]. Since then, the knowledge on microservice architectures has increased. This is especially true for microservice reference architectures (MSRAs) that serve as blueprint for other applications in the same domain. Multiple studies have been supported by the growing insight into such architectures, the coupling of the affected services, and the messaging patterns between them, as well as the resulting runtime characteristics. For instance, six reference architectures and use cases were analysed for service dependencies and interchangeability in software product lines, including a detailed study of the demonstration microservices of the Google Cloud Platform [3]. A well-known reference architecture for e-commerce is the Sock Shop, despite not having been formally introduced in the literature, due to its popularity in diverse software modernisation trainings. It consists of six polyglot backend services and one queue and is publicly available on the web[2]. Sock shop has been used among other works by the authors of MicroRCA, a root cause analysis framework to spot performance issues in microservices [30].

More focused and domain-specific understanding of the runtime properties of applications become feasible when further MSRAs are investigated and published. In recent years, this has increasingly been the case. For instance, MSRA for measurement systems and enterprise measurement infrastructures were designed and evaluated [28]. Such systems collect business metrics along with business insights and feed them into enterprise dashboards. Advanced measurement systems include semantic measurement models with distributed data management [5]. The SAMSP platform for self-adaptive microservices has been similarly published. It supports instance overload, unreachable services and other events to adjust service delivery [17]. A multi-tenant SaaS hosted in a cloud based on microservices has been validated by transforming Microsoft MusicStore [24]. A platform based on ProteomicsDB has been built to investigate the use of microservices for proteomics and personalised medicine [23]. Elasticity research in document management systems and the the investigation of service level objective violations in such elasticity scenarios yielded additional MSRAs [18, 25]. Even combined reference architectures for microservice- and blockchain-based applications have been proposed and studied [29].

While the existing MSRAs cover many domains, they are generally limited to business applications with a small or undefined set of users. Moreover, as noted by the authors of one of the elasticity works [25], they are often not built on state-ofthe-art technologies, and lack a description of asynchronous communication patterns and adaptation mechanisms that are essential for achieving scale especially in overload situations. Following the increasing digitalisation and digital transformation in society, nation-scale applications are emerging and thus warrant further analysis concerning their realisation based on microservices. This concerns in particular e-government applications addressing all citizens, residents and visitors to a country within short time periods, demanding massive elastic scalability in compute services but also adequate application design to reflect such computational capabilities.

In this article, our open source ZVAX implementation [10] shall serve as exemplary nation-scale application for the domain of digital pandemic management, and as realisation of a corresponding RA. The work consolidates our preliminary research conducted in previous years [11, 9] and adds more details, in particular a performance analysis related to deployment in local and public container platforms, and a generalisation of the architecture towards concerns found in other e-government applications. We consider ZVAX to benefit from

[2] Sock Shop website: `https://github.com/microservices-demo/microservices-demo`

a reference architecture for such applications due to the careful design and realisation of the underlying microservices. It is a fully functional system covering multiple pandemic management fields with separation of concerns along service boundaries. Hence, we argue that ZVAX serves as practical study object in microservice-related education, such as courses in software architecture or cloud-native application development. In the next sections, we first give a background on the emergence of nation-scale applications across several areas but with emphasis on e-health and pandemic management. Then, we derive the MSRA methodologically, present the microservice concepts of ZVAX and explain the realisation of the underlying MSRA on the implementation level. Moreover, we evaluate its combination of harmonic scalability and selective decentralisation. Finally, we motivate further research directions.

2 Background: Nation-Scale Applications

We define the term nation-scale application to refer to any software application that needs to scale to the population of a country, not necessarily in terms of concurrent users but within upper time boundaries that are often mere minutes. Often, these applications are in the domain of e-government addressing people, especially G2C (government to citizens) or G2P (government to citizens, residents, visitors and the general public). Evidently, there are commercial applications from private operators with even larger, de-facto global reach, but in those there are no critical moments when all users need to be reached within a small bounded time window. There are also larger applications not involving users based on machine communication (M2M, M2G), but due to being non-interactive, moderate delays are less critical and do not end up in emotionally driven overload amplification out of fear or panic. Our definition is the first to apply to software applications, whereas we would like to point out that nation-scale systems engineering has been investigated before concerning waste management, building modelling, land surveys and similar engineering concerns [20, 4], as well as batch data processing [19].

Nation-scale applications require a high-bandwidth, low-latency communication channel for broadcasting and point-to-point messaging. They are particularly linked to mobile devices, mobile telecommunications and cloud infrastructure due to the ability to trigger spike-capable access to the application within a short time period. Cell broadcast mechanisms are available and have been investigated for cloud integration [16] but are confined to non-personalised content distribution such as general information to the population in multimedia formats. Fig. 1 expresses the relationship between infrastructure, devices and applications. In the following, the characteristics of four nation-scale application areas – emergency alerts, connection inquiries, crisis expert search and finally pandemic management – are explained to determine commonalities for any nation-scale MSRA. Empirical evidence is collected from the respective Swiss and European population. The pandemic management application area is furthermore discussed in detail to give sufficient insights into the domain chosen for our specific MSRA.

The first area for applying nation-scale is in public safety, in particular emergency alert applications. A recent comparative study has compared several of those in Europe [15]. The study omitted a technical analysis of how scale is achieved on the software side but gives insight into the scale needed for crisis and hazard communication. It also documents how location-based SMS and cell broadcast is used for crisis communication, given that a false positive (a person receiving a message in error) is more tolerable than a false negative (an intended recipient not being informed in time). AlertSwiss, the emergency alert mobile application for Switzerland and Liechtenstein, has disseminated around 1200 messages over

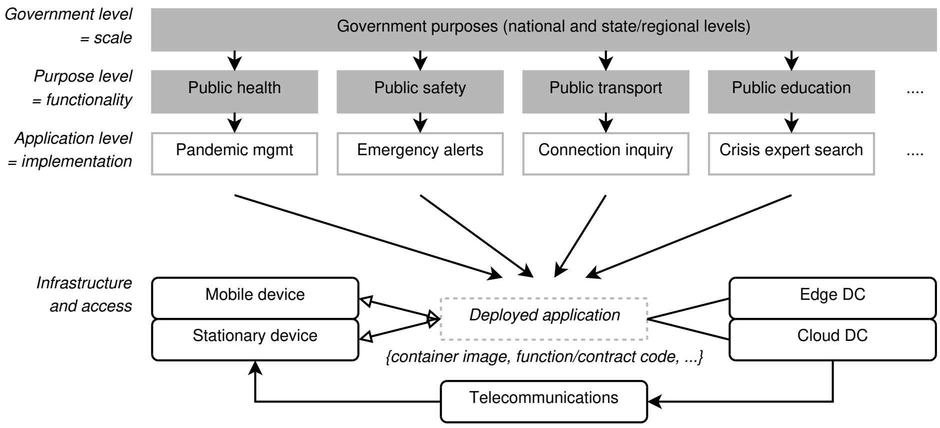


Figure 1 Infrastructure positioning of nation-scale application areas.

five years, and is supposedly reaching around 12% of the population, bounding the nation-scale level at one million people within a timeframe of second to minutes for severe notifications such as earthquakes.

The second area is in public transport, in particular connection inquiry applications. Such applications are among the most-used ones by the population due to the everyday importance of mobility. In countries providing an integrated national transport system, the acceptance and adoption is particularly high by also reaching risk-averse passengers [13]. According to a survey conducted among our students, about 94% occasionally or regularly use the timetable updates provided by the Swiss public transport system. The application is running as containerised workload across two data centres, without public numbers on associated cost or scalability characteristics. The nation-scale level can be assumed to be several million people with however a larger time window, often in the order of hours for trip planning or discounted ticket sales.

The third area is in the public education system, in particular the search of experts on a national level in crisis situations. While today, most education and training certificates are handed out on paper or as human-readable electronic documents, the trend towards microcredentials [21] is demanding new software architectures to handle automated fine-grained creation and verification of those credentials, and complex expert search and skills matching microservices on top of this information base. The Swiss public university system in its foundational programmes alone produces around 60000 certificates covering around 2 million grades which can be estimated to be broken down to more than 10 million microcredentials per year with clear peak service times tied to the semester schedules but also to crisis hiring situations.

The fourth and more deeply investigated area is in public health, in particular pandemic management, which is related to highly sensitive personal data and therefore only few messages, such as general information about pandemic spread, can be considered for broadcasting. During the COVID-19 pandemic, pandemic management services addressing the entire population were provided but often at high cost for raw data centre resources such as virtual machines, without reconsidered and rethought software architecture. In other instances, scalability problems such as downtimes and long delays occurred and made it into the national press, contributing to the population's anger about the political management of the pandemic[3]. Modern public cloud service models are promising high scalability and appear

[3] Swiss Radio & Television, Espresso, January 15, 2021

to be a solution. They are nevertheless only a part of a solution. First, public cloud services are often also limited to few thousand concurrent instances, which is insufficient for the problem field where tens of thousands of parallel requests may arrive. Second, they provide the heavy lifting but require domain-specific glue logic and interaction patterns, an area where much of the mistakes in architecture design may occur. The blueprint knowledge on how to build nation-scale microservice applications in the e-health and pandemic management area is thus limited to few approaches on vaccination passports [2, 12, 8].

3 Related Work: Reference Architectures and Benchmarks

Beyond the application area of pandemic management, software engineering research specifically aiming at the composition of applications from microservices has produced a wide range of reference architectures, both abstract ones and others applicable to particular domains. As previously mentioned, this encompasses for instance building modelling, land surveys and batch data processing [20, 4, 19] among others. In order to validate these architectures, in many cases generic experiments and benchmarks have been used by the authors, while in parallel, the research community has also contributed microservice-specific benchmarks.

Among those benchmarks are μ-Suite, Acme Air, DeathStarBench and HydraGen. μ-Suite investigated low latency applications such as image similarity or recommender systems [26] and exposed the need for latency-aware scheduling on the OS level to avoid high tail latencies. Acme Air was originally a monolithic web performance benchmark. It was adapted to Node.js and Java microservice execution [27] with credible reports about the performance dropping to around 20% of the monolithic version due to overheads. DeathStarBench incorporates sample applications such as social networks, media and e-commerce sites, online banking and vehicle control [14]. It is available as open source framework and again reports tail latencies at scale among other metrics. Its authors shifted the emphasis from a comparison against monolithic architectures towards a reasoning about certain performance quirks. HydraGen takes benchmarks a step further and generates them [22]. HydraGen has been validated in traffic engineering but its design does not preclude other application domains.

Hence, it becomes apparent upfront that a pure performance-oriented design might not be best served by microservices. However, elastic scalability beyond machine boundaries and the increasing investments of cloud providers into microservice hosting platforms suggest that the drawbacks might become less significant especially for nation-scale requirements. Consequently, researchers have investigated scaling and scalability properties such as by-design global scaling [1], but also orthogonal concepts such as API patterns including scalability-related quality patterns [31] and deployment [6].

4 Concepts: Scalable Pandemic Management

We condense the problems raised in the introduction into a single research problem: Which software design and architecture adequately fits the e-health domain and more specifically the task of nation-scale pandemic management? In order to address the problem, we follow a methodology consisting of four steps. In the first methodological step, we contribute four novel concepts as generic, domain-independent architectural design foundation.

1. Nation-scale services. They are built to serve large parts of a population in a short amount of time. In quantitative terms, we assume a lower bound of many 10,000s of short-lived requests per second. This purposefully exceeds current public cloud offerings by an order of magnitude even when combining regions. We note that the target concurrency

can often be influenced by political means outside the technical scope, for instance by population segmentation by age group. Such segmentation may however not be applicable in emergency situations. Under no circumstances should users perceive downtime or unresponded hanging requests, due to the danger of exacerbating the system load by follow-up actions in panic.

2. Harmonic scalability. A distributed system is harmonically scalable if all of its constituent parts scale along the critical request paths. For an architecture based on microservices, this entails the ability to scale elastically in each service, but also in attached middleware services. In geometric terms, the system representation may overall shrink or grow but the proportions remain the same.
3. Selective decentralisation. The scalability is influenced by user preferences on where to store and process data. Hence, decentralisation is used to reduce microservice invocation load at neuralgic points while at the same time offering stronger privacy guarantees on demand. From an architectural perspective, the selective decentralisation leads to a selectively externalised statefulness, referring to the state as output of one microservice that determines the follow-up behaviour of another microservice.
4. “Flatten the curve”. This term originating in the domain-specific goals of pandemic management also applies to the prevention of microservice overload. Queues and other asynchrony mechanisms are used to facilitate quick responses to service requests, leaving the bulk of the work for less loaded periods of time while granting a rapid responsive behaviour to users navigating the user interface. Users are then informed asynchronously at later points in time about the results of compute-centric tasks through notification channels depending on the registered contact details.

Next, we derive a domain-specific reference architecture for the domain of pandemic management. The architecture must support the functional requirements implied by the domain, and adapt to the underlying infrastructure especially in terms of computation and communication. A representative application covering the main governmental interests in pandemic management consists of the following four classes of services to the population.

1. Appointments. In order to prevent people from queueing unnecessarily, appointments help to “flatten the people curve” in real-world situations such as testing and vaccination points.
2. Certificate creation and signing. Various schemes exist, with most having settled on a QR code representation digitally signed by a single authority or, for better fraud protection, multiple authorities.
3. Certificate checking. The inverse process, combining signature validity checks with expiration policies.
4. Information solicitation. This encompasses mandatory solicitations such as passenger locator forms before arriving in a country (triggering a certificate creation) and contact quarantine tracing upon cases of assumed or confirmed infections, but also voluntary information provided to assist the tracing.

These four classes of services need to be mapped to four or more technological realisations. In a third methodological step, we therefore combine the conceptual requirements and functional scope, and match them against recent technological progress. This means that all the management services of the four classes are instantiated and delivered with high elastic scalability and low latency in order to achieve the desired volume of requests. Four main technological advances are increasingly available in managed microservice environments and are considered favourable from the reference architecture perspective.

1. Multi-core function runtimes and container-native hosting for maximised local parallelism, attached to low-latency local storage such as RAM and NVRAM. Often, the underlying execution follows a uniform model based containerisation or hardware-accelerated virtualisation such as ARM TrustZone/vTZ, AMD-V or Intel VT-x. To the software engineer, the execution offers deployment interfaces as raw container images (following the de-facto standard set by the Open Container Initiative), function source code (FaaS with its many language- and provider-specific syntax requirements and constraints), source code in microservice-oriented languages (e.g. Jolie), or smart contracts in managed blockchains (e.g. canisters). All non-image software artefacts are automatically converted to appropriate images upon deployment, keeping the engineer free from infrastructural concerns but also limiting potentially performance- or latency-improving tuning. We consider all of those interfaces valid technological choices for a microservice design as long as service-oriented characteristics (well-defined interface supporting one or more message exchange pattern through loose coupling) are fulfilled. Containers, in particular, do not have an inherent service orientation but are suitable to encapsulate one or multiple services.
2. Event-driven asynchronous function execution. Microservice instances, in particular when designed as event-triggered functions, scale almost proportional to the request rate. Factors such as cold start (for initial invocation) or spawn start (for concurrent invocation) have been thoroughly investigated in recent years and are now well understood, and with methods such as prewarming and idle time optimisation, further scalability gains can be achieved.
3. Preparedness for extreme edge deployments. New hardware allows for deploying microservices directly on network interfaces (e.g. SmartNICs) in edge data centres, allowing for unprecedented scaling of stateless services by geographic distribution close to the points of use and their low-latency delivery. This can be exploited for instance to distribute the work-intensive QR code creation and signing processes that require no state other than a set of input parameters. Moreover, federated deployments become possible to map political hierarchies and pandemic management responsibilities (e.g. federal-level, state-level) to delivery locations. At the same time, the system needs to remain deployable on a single off-the-shelf device to achieve full elasticity from single developer to nation-scale.
4. Flexible bindings to communication infrastructure. For instance, a service can bind to telecommunication cloud services to obtain a regional cell broadcasting interface, or to satellite providers for full global coverage but with narrowband links. Alternatively, it can use conventional mobile backend-as-a-service (MBaaS) interfaces for personalised push notifications. In conjunction with edge deployments, these topological concerns are of interest when considering nation-scale beyond the mainland boundaries of countries, in particular for island nations or when governments want to address their citizens abroad. In a Swiss context, around 9% of the population or 800000 citizens live abroad and might need to be included in short-lived e-government processes such as electronic voting, but also in long-term services that benefit from follow-the-sun microservice provisioning semantics across federated clouds or edges.

The concrete management system reference architecture results as a fourth step of the methodology. It is synthesised along with constituent microservices in Fig. 2, starting from the user perspective of either personal access through a mobile device or computer, or a stationary device such as a QR code scanner located at the entrance of a location that mandates such checks – such as restaurants, public authorities or university campuses.

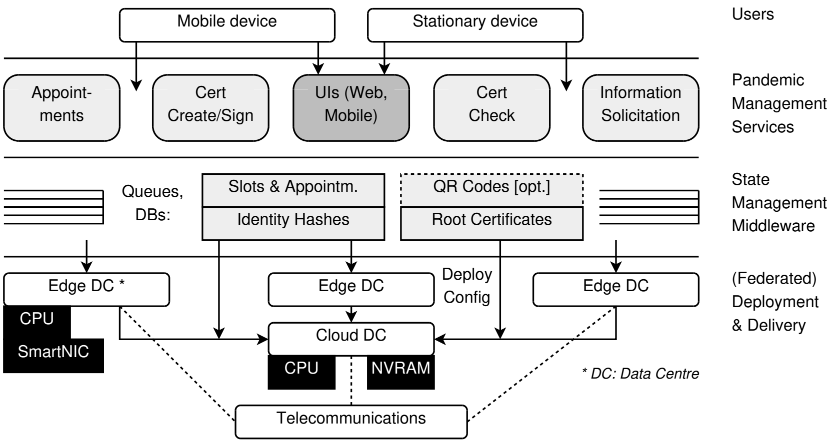


Figure 2 Reference architecture; components marked [opt.] are optional and only activated on demand depending on user preferences due to being offloadable to client-side devices.

The architecture reflects the ability to choose, on a per-user basis, whether to opt into more government-managed or more self-managed data records, with corresponding responsibilities for backups and access protection. Moreover, it aligns with the increasing availability of micro datacentres and other edge deployments for scalable service delivery close to the users, as well as different capabilities in telecommunications infrastructure, including selective availability of cell broadcast and often still limited access to such facilities from cloud APIs.

5 Implementation: ZVAX

ZVAX, the Zurich vaccination and pandemic management software [10], is a fully functional demonstration system of the MSRA for nation-scale pandemic management. Its development started in the second year of the COVID-19 pandemic, and its functional extensions were driven by digital management processes introduced by governments around the world, with emphasis on the services on different administrative levels of the Swiss confederation, Northern Macadonia, and other European countries. Hence, apart from implementing the MSRA and thus overcoming scalability and privacy issues, it also serves as testbed for solving challenges that emerged on a societal level in Switzerland, often with extensive local or national press coverage. These challenges relate to many cases of certificate fraud (solved in ZVAX by multi-signatures, leading however to larger QR codes), interoperability (solved by multi-QR code schemes), different expectations on privacy (solved by selective decentralisation), and flexible federated deployment (solved by an adaptive user interface connected to configurable sets of backend services). ZVAX can furthermore serve as testbed for improved client-side QR code detection, for instance on black background that usually causes problems in today's mobile applications, and coloured or animated codes to represent portrait photos for safer identification.

The microservice architecture of ZVAX is shown in Fig. 3. The figure omits the information solicitation service to focus on those of relevance for the most complex workflow: making an appointment for vaccination, creating and signing a certificate during vaccination, and checking the certificate afterwards.

Fig. 4 gives an impression of the user-facing functionality of the current implementation. Users are able to select the desired level of privacy, in turn determining the degree of centralised versus decentralised data storage. Decentralisation is achieved by a combination

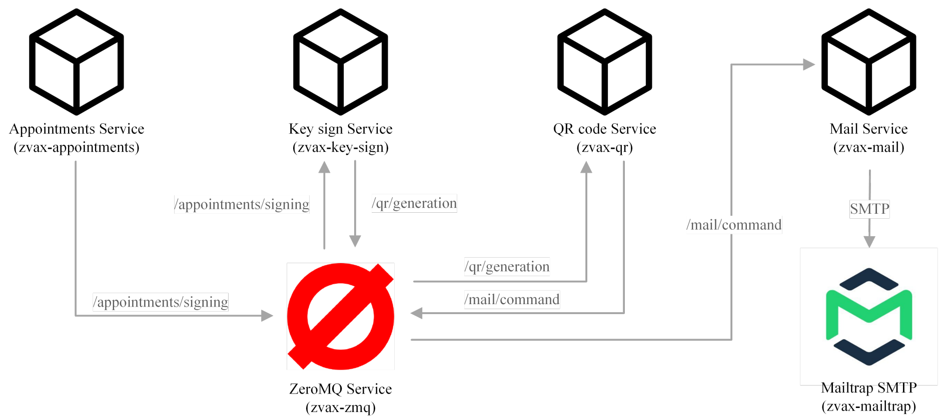


Figure 3 Microservice composition of the ZVAX implementation.

of in-browser storage and file downloads, leading to no trace of any health-related activities when full decentralisation is selected by the user. Subsequently, users apply for a test or vaccination appointment, retrieve certificates, enter locator form information, or manage contact tracing. Correpondingly, doctors and health officials are able to sign (and counter-sign) as well as verify certificates.

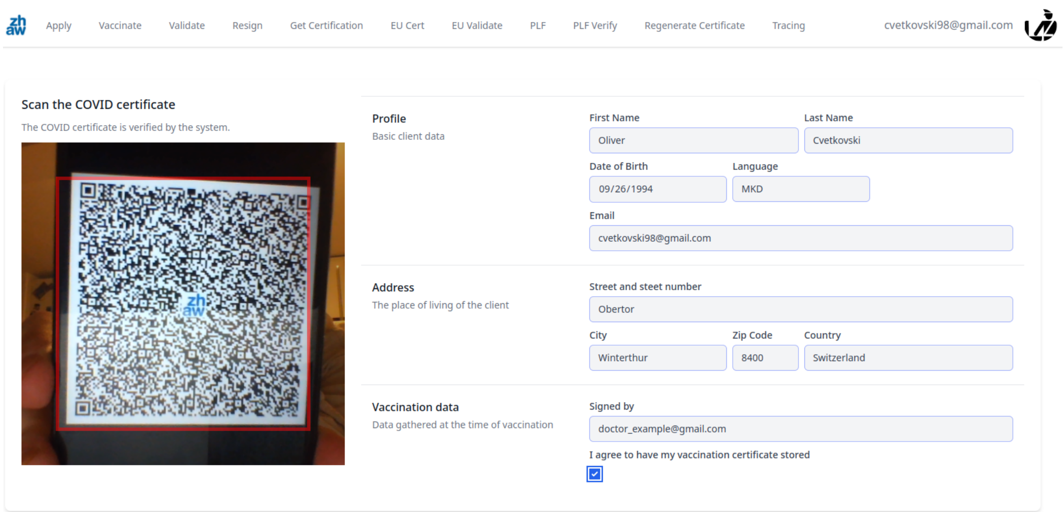


Figure 4 Exemplary screenshot of the QR code verification within the ZAX implementation.

Appointments are possible for groups of persons, reducing the need to fill out forms considerably for families. The appointments-related invocation flow on the microservice level encompassing the above-introduced microservices, not including the middleware components, is shown in Fig. 5. All microservices can be scaled with instance selection through the Traefic load balancer. First, the authentication service grants a time-limited token. Then, the appointments service is invoked with the token. In case an appointment can not be obtained immediately or the system is overloaded, an asynchronous re-invocation is scheduled ("flatten the system load curve"). Next, the appointment is expressed as a signed QR code, re-using the same services that will also perform the same task for vaccination certificates.

All personal information apart from the time slot blocking and an identity hash are then removed from the system, and the QR code is sent via e-mail for the person(s) having received an appointment.

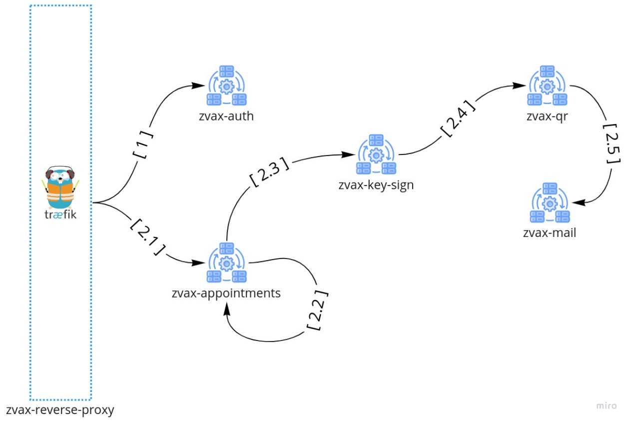


Figure 5 Exemplary microservice invocation flow consisting of two subflows and six steps.

The ZVAX implementation is easy to bootstrap on new machines due to its complete containerisation. The codebase is prepared to be scaled horizontally with orchestrators such as Docker Compose, Docker Swarm and Kubernetes. Almost all microservices ship with a single implementation. In order to stress the substitution principle and to compare polyglot implementations, the QR code microservice ships with two implementations, in Python and in Rust. This has not only performance implications, but also affects the scaling behaviour due to differences in size and resource requirements of the resulting container images. We have carefully designed both the individual microservices and the entire composition to adhere to harmonic scaling. Often, this term refers to harmonic local-versus-global scaling in the general sense [1] whereas we specifically followed the request path and ensured that no bottleneck would occur that could starve subsequent processing steps along the same path.

6 Evaluation: Performance and Scalability

We have evaluated ZVAX to demonstrate its preparedness for nation-scale deployments, primarily by taking performance and correctness measurements at different active user scale levels. To compare modern cloud-native microservice deployments including Function-as-a-Service (FaaS) and auto-scaled Kubernetes instances, we have set up a scalable testbed on large virtual machine instances on the OpenStack cluster 'apu' in Zurich University of Applied Sciences. The testbed consisted of a single master node and three worker nodes, from which the results of larger deployments can be interpolated. ZVAX got deployed to the infrastructure via Helm. In addition to the system under test, the cluster hosts a set of auxiliary services to better diagnose runtime results. This includes a monitoring stack using Grafana and Prometheus, an alternate autoscaler called KEDA (Kubernetes Event-Driven Autoscaling), OpenFAAS and Cert-Manager. Grafana and Prometheus provide dashboards that display metrics about the cluster, K6 tests, Keda and RabbitMQ. Cert-Manager is used to provision certificates (related to the infrastructure communication, not related to pandemic management) automatically. The entire setup is shown in Fig. 6.

Spike and stress testing experiments are done by using K6 by Grafana. The tool allows us to design detailed scenarios that are executed onto the chosen service endpoints via simple HTTP requests. Prometheus is used as a monitoring solution to analyse how the services

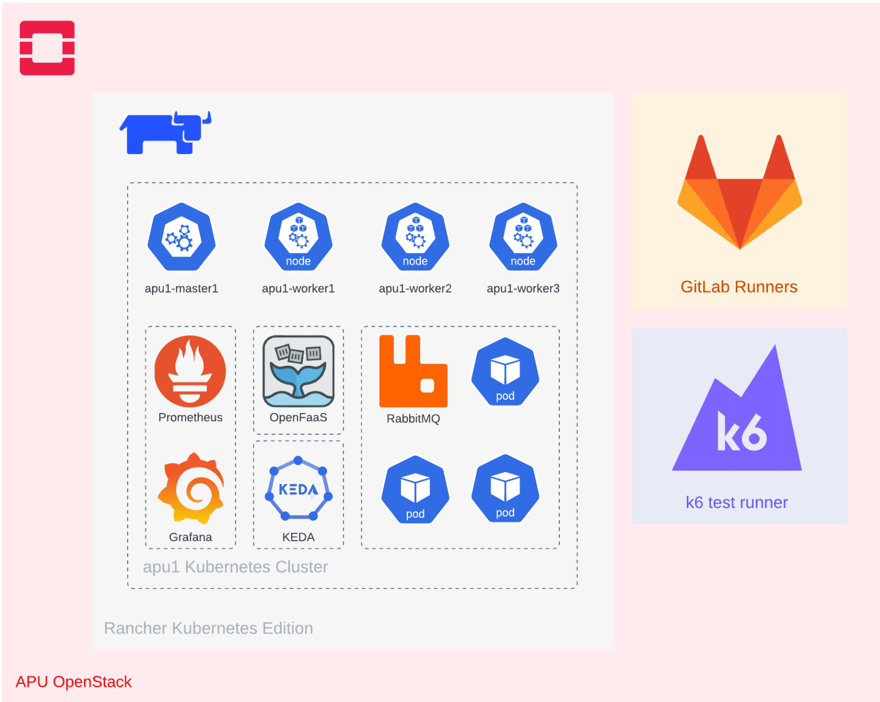


Figure 6 Testing and evaluation infrastructure.

behave regarding CPU and memory consumption. Tests are per service and endpoint. Each load test is tuned to that specific service, as not all services have the same load and demands in a real-world scenario. Estimated loads are determined using available COVID-19 datasets and both measured and estimated computational complexity. Tests are run using variable scaling. Service replica count and CPU limits can be tuned for each test using Helm. The evaluation aims to answer three main questions: (i) How many resources does an individual service instance require?; (ii) How do we combine variables to achieve optimal scalability?; (iii) Where are bottlenecks? Load profiles are established using estimated load, ranging from 0.25x (tier 1) to 2x (tier 6). The experiments encompass the time required for QR code generation, certificate signing, certificate storage, appointment booking, as well as alternative QR code generation through OpenFAAS, the Rust-based implementation and KEDA, and alternative appointment booking through connection pooling.

Due to limited space, not all experiments can be replicated here. We include two interesting results that show the benefits of the chosen concept. First, the appointment booking with connection pool of size 20 with a buffer of 40 (Fig. 7) offsets a high error rate that is otherwise introduced by autoscaling, by harmonising request queueing and processing. The mean failure rate drops from around 4% to around 0.10%.

Second, the QR code was generated following the Swiss distribution of PCR tests with a peek rate of 2.48 per second and vaccination/booster with a peek rate of 2.08 per second, totalling 4.56 requests/s, and when assuming an eight-hour work window for vaccination and test centres, 6.84 requests/s (RPS). Fig. 8 compares the median HTTP request duration for (a) static replica counts and (b) horizontal autoscaling (HPA). On tier 3 as comparison point, autoscaling achieved 7 RPS, sufficient for the peek rates, with a replica CPU limit

	Appointments Service (DB Connection Pooling) (/)					
Autoscaling	Failure rate		Median HTTP request duration on success [ms]		p(95) HTTP request duration on success [ms]	
Request/s	12.19	16.25	12.19	16.25	12.19	16.25
Req/s rounded up	13.00	17.00	13.00	17.00	13.00	17.00
100m	0.10%	7.86%	3708	4798	17401	20227
250m	0.02%	0.01%	32	42	1015	2092
500m	0.03%	0.05%	29	31	1397	3077
1000m	0.05%	0.03%	36	148	4872	8070
1500m	0.02%	0.02%	169	595	8083	10043

Figure 7 Autoscaled and connection-pooled appointment booking.

of 1000 millicores, preventing the system from throttling. HPA is a reactive autoscaler based on deferred metrics (metrics server interval + HPA interval + scale-up reaction time), leading to insufficient reaction agility in contrast to anticipating autoscalers. With KEDA, the metrics resolution issue disappears but the remaining slowness, in particular the 15 seconds HPA interval, remains. Moreover, as the chosen scaling setting for KEDA is based on the queue length, sudden bursts of traffic may cause the queue length to climb drastically, as instances cannot keep up with queue intake. In such cases, KEDA may overprovision instances until the system stabilises. As a conclusion from this experiment, we can consider it a positive coincidence that HPA provided sufficient upscaling speeds for the observed peek rate development, but for safe-guarding similar scenarios in which anticipation is not possible, more rapidly reacting autoscalers and pod schedulers should be developed for Kubernetes.

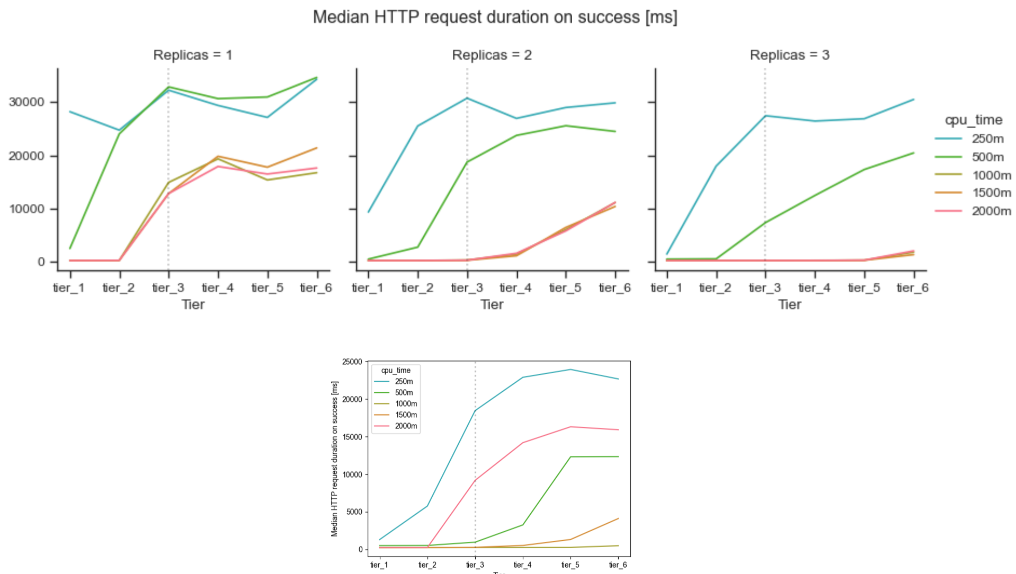


Figure 8 Static and autoscaled QR code generation.

Overall, our architecture has shown to be sufficiently scalable for a country the size of Switzerland on 4 VMs, but required tuning and exploring the different deployment options. The influence of deployment is therefore considered as important as the influence of the microservice design.

A crucial and open discussion point is the generalised design of such software architectures for deployments in arbitrary countries, including those with approximately 100 times the population size. Apart from the mentioned geographic topologies (edge computing, federated cloud or multi-cloud), such designs will likely benefit from hierarchical structures following the administration levels, such as states and union territories in India or provinces, autonomous regions and direct-administered municipalities in China. Fiduciary agreements can then be used to balance operational effort and scalability needs, for instance, by a single cloud deployment covering several smaller states or provinces.

7 Conclusions and Future Work

With the open source system ZVAX[10], we have developed a pandemic management system that improves upon current government-provided services. It is designed to accomodate surge requests and sustained requests at a higher scale, and adjusts to user preferences concerning data handling through selective decentralisation. ZVAX builds upon a domain-specific MSRA that takes current technological progress into account and lets researchers explore more flexible externalised state management beyond the conventional stateful/stateless distinction. Moreover, ZVAX is positioned as study object for the training of software engineers who learn the construction of societally relevant digital services.

Future work is divided into domain-specific and general architectural directions. Within the domain of pandemic management systems, we aim at reducing the need for complementing the QR code with a passport by embedding photographic identity information within a static (coloured) or animated code. On the architectural level, we aim at conducting large-scale and nation-scale performance tests to prove the proposed approach at all levels: services, queues, composition and deployment. This will encompass the acceleration gained by edge deployments, and thus contribute to the debate on digital souvereignty of administrative levels achievable through a wider digital transformation enabled by flexibly deployable microservices.

References

1 Lorenzo Bacchiani, Mario Bravetti, Maurizio Gabbrielli, Saverio Giallorenzo, Gianluigi Zavattaro, and Stefano Pio Zingaro. Proactive-reactive global scaling, with analytics. In Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés, editors, *Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proceedings*, volume 13740 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2022. `doi:10.1007/978-3-031-20984-0_16`.

2 Masoud Barati, William J. Buchanan, Owen Lo, and Omer F. Rana. A privacy-preserving distributed platform for COVID-19 vaccine passports. In Luiz F. Bittencourt and Alan Sill, editors, *UCC '21: 2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing, Leicester, United Kingdom, December 6 - 9, 2021 - Companion Volume*, pages 16:1–16:6. ACM, 2021. `doi:10.1145/3492323.3495626`.

3 Benjamin Benni, Sébastien Mosser, Jean-Philippe Caissy, and Yann-Gaël Guéhéneuc. Can microservice-based online-retailers be used as an spl?: a study of six reference architectures. In Roberto Erick Lopez-Herrejon, editor, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 24:1–24:6. ACM, 2020. `doi:10.1145/3382025.3414979`.

4 Andy S. Berres, Brett C. Bass, Mark B. Adams, Eric Garrison, and Joshua R. New. A data-driven approach to nation-scale building energy modeling. In Yixin Chen, Heiko Ludwig, Yicheng Tu, Usama M. Fayyad, Xingquan Zhu, Xiaohua Hu, Suren Byna, Xiong Liu, Jianping Zhang, Shirui Pan, Vagelis Papalexakis, Jianwu Wang, Alfredo Cuzzocrea, and Carlos Ordonez, editors, *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, pages 1558–1565. IEEE, 2021. `doi:10.1109/BIGDATA52589.2021.9671786`.

5 Eric Braun. *Microservice-based Reference Architecture for Semantics-aware Measurement Systems*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2020. URL: https://nbn-resolving.org/urn:nbn:de:101:1-2020112503582180757762.

6 Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. Optimal and automated deployment for microservices. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2019. doi:10.1007/978-3-030-16722-6_21.

7 Antonio Brogi, Andrea Canciani, Davide Neri, Luca Rinaldi, and Jacopo Soldani. Towards a reference dataset of microservice-based applications. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*, volume 10729 of *Lecture Notes in Computer Science*, pages 219–229. Springer, 2017. doi:10.1007/978-3-319-74781-1_16.

8 Andreea Ancuta Corici, Tina Hühnlein, Detlef Hühnlein, and Olaf Rode. Towards interoperable vaccination certificate services. In Delphine Reinhardt and Tilo Müller, editors, *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, pages 139:1–139:9. ACM, 2021. doi:10.1145/3465481.3470035.

9 Oliver Cvetkovski, Carlo Field, Davide Trinchi, Christof Marti, and Josef Spillner. ZVAX - A Microservice Reference Architecture for Nation-Scale Pandemic Management. International Conference on Microservices (Microservices) – Extended Abstract, may 2022.

10 Oliver Cvetkovski, Carlo Field, Davide Trinchi, and Josef Spillner. ZVAX - Zurich vaccination and pandemic management software. Zenodo, apr 2023. doi:10.5281/zenodo.7869751.

11 Oliver Cvetkovski and Josef Spillner. A self-contained, decentralized and nation-scale approach to e-government services. 10th IEEE International Conference on Cloud Computing in Emerging Markets (CCEM) – Student Project Showcase, oct 2021.

12 Mauricio de Vasconcelos Barros, Frederico Schardong, and Ricardo Felipe Custódio. Leveraging self-sovereign identity, blockchain, and zero-knowledge proof to build a privacy-preserving vaccination pass. *CoRR*, abs/2202.09207, 2022. arXiv:2202.09207.

13 Anthony Downward, Subeh Chowdhury, and Chapa Jayalath. An investigation of route-choice in integrated public transport networks by risk-averse users. *Public Transp.*, 11(1):89–110, 2019. doi:10.1007/S12469-019-00194-0.

14 Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 3–18. ACM, 2019. doi:10.1145/3297858.3304013.

15 Andrin Hauri, Kevin Kohler, and Benjamin Scharte. A comparative assessment of mobile device-based multi-hazard warnings: Saving lives through public alerts in europe. *CSS Risk and Resilience Reports*, 2022.

16 Michail-Alexandros Kourtis, Begoña Blanco, Jordi Pérez-Romero, Dimitris Makris, Michael J. McGrath, George Xilouris, Daniele Munaretto, Ruben Solozabal, Aitor Sanchoyerto, Ioannis Giannoulakis, Emmanouil Kafetzakis, Vincenzo Riccobene, Elisa Jimeno, Anastasios Kourtis, Ramon Ferrús, Fidel Liberal, Harilaos Koumaras, Alexandros Kostopoulos, and Ioannis P. Chochliouros. A cloud-enabled small cell architecture in 5g networks for broadcast/multicast services. *IEEE Trans. Broadcast.*, 65(2):414–424, 2019. doi:10.1109/TBC.2019.2901394.

17 Peini Liu, Xinjun Mao, Shuai Zhang, and Fu Hou. Towards reference architecture for a multi-layer controlled self-adaptive microservice system. In Óscar Mortágua Pereira, editor, *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018*, pages 236–235. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018. `doi:10.18293/SEKE2018-086`.

18 Manuel Ramírez López and Josef Spillner. Towards quantifiable boundaries for elastic horizontal scaling of microservices. In Ashiq Anjum, Alan Sill, Geoffrey C. Fox, and Yong Chen, editors, *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 35–40. ACM, 2017. `doi:10.1145/3147234.3148111`.

19 Sahar Mazloom, Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Secure parallel computation on national scale volumes of data. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2487–2504. USENIX Association, 2020. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/mazloom`.

20 William J. B. Midgley, Michael J. de C. Henshaw, and S. Alshuhri. A systems-engineering approach to nation-scale problems: Municipal solid waste management in saudi arabia. *Syst. Eng.*, 24(6):480–496, 2021. `doi:10.1002/SYS.21597`.

21 Laurie Pickard, Dhawal Shah, and J. J. De Simone. Mapping microcredentials across MOOC platforms. In *Learning With MOOCS, LWMOOCS 2018, Madrid, Spain, September 26-28, 2018*, pages 17–21. IEEE, 2018. `doi:10.1109/LWMOOCS.2018.8534617`.

22 Mohammad Reza Saleh Sedghpour, Aleksandra Obeso Duque, Xuejun Cai, Björn Skubic, Erik Elmroth, Cristian Klein, and Johan Tordsson. Hydragen: A microservice benchmark generator. In *16th IEEE International Conference on Cloud Computing, CLOUD 2023, Chicago, IL, USA, July 2-8, 2023*, pages 189–200. IEEE, 2023. `doi:10.1109/CLOUD60044.2023.00030`.

23 Marwin Shraideh, Patroklos Samaras, Maximilian Schreieck, and Helmut Krcmar. A microservice-based reference architecture for digital platforms in the proteomics domain. In Leona Chandra Kruse, Stefan Seidel, and Geir Inge Hausvik, editors, *The Next Wave of Sociotechnical Design - 16th International Conference on Design Science Research in Information Systems and Technology, DESRIST 2021, Kristiansand, Norway, August 4-6, 2021, Proceedings*, volume 12807 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2021. `doi:10.1007/978-3-030-82405-1_26`.

24 Hui Song, Phu Hong Nguyen, Franck Chauvel, Jens M. Glattetre, and Thomas Schjerpen. Customizing multi-tenant saas by microservices: A reference architecture. In Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama, editors, *2019 IEEE International Conference on Web Services, ICWS 2019, Milan, Italy, July 8-13, 2019*, pages 446–448. IEEE, 2019. `doi:10.1109/ICWS.2019.00081`.

25 Sandro Speth, Sarah Stieß, and Steffen Becker. A saga pattern microservice reference architecture for an elastic SLO violation analysis. In *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*, pages 116–119. IEEE, 2022. `doi:10.1109/ICSA-C54293.2022.00029`.

26 Akshitha Sriraman and Thomas F. Wenisch. μ suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*, pages 1–12. IEEE Computer Society, 2018. `doi:10.1109/IISWC.2018.8573515`.

27 Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*, pages 85–94. IEEE Computer Society, 2016. `doi:10.1109/IISWC.2016.7581269`.

28 Matthias Vianden, Horst Lichter, and Andreas Steffens. Experience on a microservice-based reference architecture for measurement systems. In Sungdeok (Steve) Cha, Yann-Gaël Guéhéneuc, and Gihwon Kwon, editors, *21st Asia-Pacific Software Engineering Conference,* *APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 183–190. IEEE Computer Society, 2014. doi:10.1109/APSEC.2014.37.

29 Yanze Wang, Shanshan Li, Huikun Liu, He Zhang, and Bo Pan. A reference architecture for blockchain-based traceability systems using domain-driven design and microservices. *CoRR*, abs/2302.06184, 2023. doi:10.48550/ARXIV.2302.06184.

30 Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020. doi:10.1109/NOMS47738.2020.9110353.

31 Olaf Zimmermann, Mirko Stocker, Daniel Lubke, Uwe Zdun, and Cesare Pautasso. *Patterns for API design: simplifying integration with loosely coupled message exchanges*. Addison-Wesley Professional, 2022.

Custom Serverless Function Scheduling Policies: An APP Tutorial

Giuseppe De Palma[1]
Università di Bologna, Italy

Saverio Giallorenzo
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Jacopo Mauro
University of Southern Denmark, Odense, Denmark

Matteo Trentin
Università di Bologna, Italy
University of Southern Denmark, Denmark

Gianluigi Zavattaro
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Abstract

State-of-the-art serverless platforms use hard-coded scheduling policies that hardly accommodate users in implementing functional or performance-related scheduling logic of their functions, e.g., preserving the execution of critical functions within some geographical boundaries or minimising data-access latencies. We addressed this problem by introducing APP: a declarative language for defining per-function scheduling policies which we also implemented as an extension of the open-source OpenWhisk serverless platform. Here, we present a gentle introduction to APP through an illustrative application developed over several incremental steps.

2012 ACM Subject Classification Software and its engineering → Scheduling; Computer systems organization → Cloud computing

Keywords and phrases Serverless, Function Scheduling, Declarative Languages, Tutorial

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.5

Supplementary Material *Software (Source Code)*: `https://github.com/giusdp/openwhisk/tree/old_app`, archived at `swh:1:dir:ba4498167bdd7b5c4a2cbc676dcc04e8df6e8354`

Funding Research partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)".

1 Introduction

Serverless is a cloud computing model that has become increasingly popular in recent years. In serverless, a provider manages the dynamic allocation of resources needed to satisfy some inbound requests (such as HTTP requests, database updates, or scheduled events), removing from the shoulders of programmers the burden of managing and scaling both the infrastructure and runtime platforms.

[1] Corresponding author

Besides easing software deployment, serverless adopts a per-usage cost model, where costs correspond to the resources used to answer requests, which ditch the expenses of running idle servers when no requests reach the system. Examples of serverless platforms include AWS Lambda [9], Microsoft Azure Functions [10], and Google Cloud Functions [24].

A common trait of these platforms is that they manage the allocation of functions over the available computing nodes, also called *workers*, following opinionated policies that favour some performance principle. Indeed, effects like *code locality* [28] – due to latencies in loading function code and runtimes – or *session locality* [28] – due to the need to authenticate and open new sessions to interact with other services – can sensibly increase the run time of functions. As a consequence, there exists ever-growing literature on serverless scheduling techniques that mix one or more of these locality principles to increase the performance of function execution [41, 37, 35, 1, 42, 50, 36, 32, 47, 44, 45, 48, 43, 16, 33, 13, 15, 30, 34, 46]. Besides performance, functions can have functional requirements that constrain the choices of the scheduler. For example, users might want to avoid allocating their functions alongside "untrusted" ones for security purposes [11, 51, 4, 18].

Albeit a FaaS platform can mix one or more principles to expand the profile coverage of function execution, platform-wide policies hardly suit all kinds of performance profiles. For this reason, in [20, 19] we started a new line of research dedicated to the introduction of a modular approach to FaaS scheduling. The approach hinges on the definition of custom, per-function policies through a domain-specific declarative language, called *Allocation Priority Policies* (APP). Thanks to APP, users can define co-existing scheduling policies, each best suited for its function (or sets thereof). In [20, 19], we validated our approach by extending the open-source Apache OpenWhisk serverless platform to support APP scripts and by showing that our extensions outperform vanilla OpenWhisk in locality-bound scenarios [20, 19].

In this paper, we present a gentle introduction to APP, by means of a tutorial. Even though APP is a platform-independent language, to exemplify its usage, we choose to describe it in the context of the OpenWhisk platform, which is both one of the most studied serverless platforms in the literature [26] and actively developed among the open-source alternatives.

We start our presentation in Section 2 by discussing the OpenWhisk serverless platform and introduce the APP language to control the scheduling of functions. To keep this tutorial at an introductory level, we consider the first, minimal version of APP, as appeared in [20]. We illustrate the language through realistic, incremental use cases in Section 3. We assume the reader to have basic knowledge of software programming, computer architectures and networks, and of cloud concepts like virtual machines and containers. The interested reader can deploy the extended OpenWhisk platform used in this tutorial by using Terraform and Ansible scripts provided at [40].

In Section 4, we discuss both language primitives found in an extended version of APP published in [19] and ongoing work on future extensions. We conclude in Section 5 by discussing related work and drawing final remarks.

2 Serverless Computing and the APP Language

We start by introducing in more concrete terms FaaS and the details of serverless platforms (relevant to function scheduling) by describing the architecture and execution model of Apache OpenWhisk. We deem these preliminaries useful to understand the constructs found in the APP language, presented afterwards.

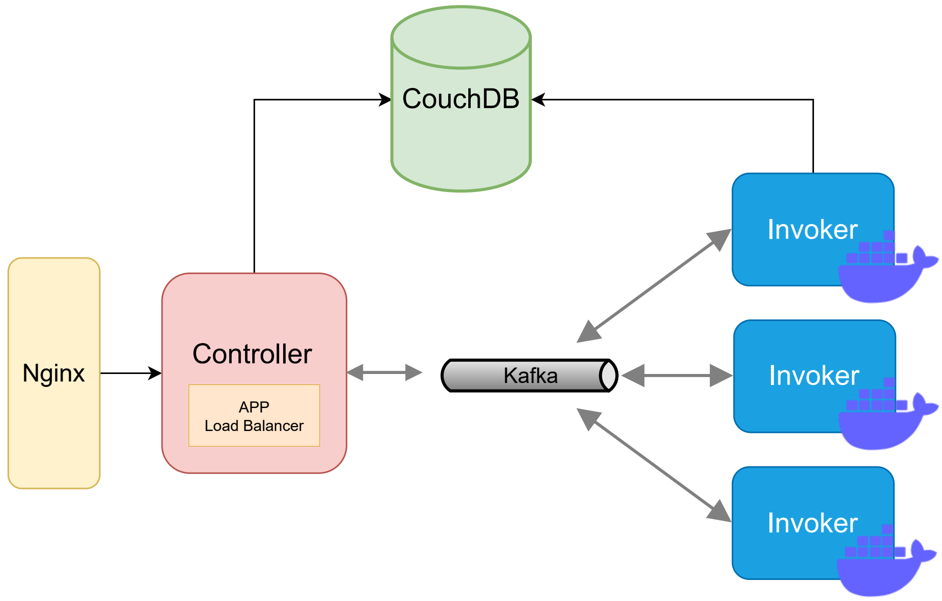


Figure 1 Diagram of the OpenWhisk Architecture with the Controller component adapted to host APP-based custom scheduling policies.

2.1 Apache OpenWhisk

Apache OpenWhisk [38] is an open-source serverless computing platform, widely-used in research and adopted by some cloud providers (IBM Cloud Functions [17] and Digital Ocean Functions [21]). In the OpenWhisk programming model, developers write "actions" which are stateless functions that run on the platform. These functions can be written in any supported programming language (e.g., Go, Java, JavaScript, PHP, Python, Ruby, Swift).

Functions can be executed by defining rules associating a trigger (e.g., an event like HTTP requests) with a certain function. When an event is received, OpenWhisk identifies the appropriate function to execute based on the event type and the function's declared trigger. Each function is executed within a container that is created and managed by OpenWhisk. The platform scales the number of containers up or down based on the workload, so developers do not have to worry about provisioning or managing infrastructure.

OpenWhisk consists of five components (see Figure 1). The entry point for requests is an Nginx reverse proxy that redirects requests to (one or more) Controllers. Controllers are the characterising component of the architecture. They manage the overall state of the system, including request handling, user authentication, and function deployment. Since they are the component that chooses which worker shall execute a given function, Controllers work also as a Load Balancers. Note that, in Figure 1, we label workers as Invoker(s), which is the nomenclature used in the OpenWhisk project to dub workers. The message broker used to communicate between Controllers and workers is Apache Kafka [6]. Workers are responsible for creating Docker containers, initialising them with functions' code, and running the functions. A CouchDB database is used to store the state of the system, including information about users, functions, triggers, rules, invocation requests, and invocation results.

Controllers use a hardcoded scheduling policy dubbed "co-prime scheduling". The co-prime logic allocates functions on workers by associating a function to a hash and a step size. The hash finds the primary worker, called the "home" worker. The step size finds a list of workers used in succession when the preceding ones become unavailable, e.g., due to overload. This routine allows OpenWhisk's scheduler to route invocations of the same function to the same worker(s), resulting in a high probability of cached container reuse. This practice minimises the occurrence of "cold starts", i.e., the increased latency in function execution due to the overhead of fetching the code of the function to put it in execution.

2.2 The APP Language

Similarly to OpenWhisk, the other FaaS platforms adopted in the industry come with hardcoded scheduling policies which may not always meet the specific needs of developers and providers. For example, OpenWhisk maximises container reuse to avoid cold starts, but it does not take into account other factors that can impact the latency and improve performance (least of all satisfy individual functional requirements on scheduling). As an example of performance improvement, let us have a function that accesses a database. Running that function on a pool of machines close to the database would reduce network traffic and latency.

The motivation behind APP is to allow a FaaS platform to follow specific scheduling policies depending on which function must be scheduled for execution. APP helps users optimise their serverless architectures by allowing them to define customised FaaS scheduling policies that instruct the platform on which workers are best suited to run each function. From the architectural point of view, letting serverless platforms support APP is straightforward and one mainly needs (as we did in our extensions [20, 19]) to modify the Controller (see Figure 1) so that it follows the scheduling policies defined in a given APP script.

Syntax-wise, APP adopts the current trend of configuration files in popular DevOps and Cloud tools (e.g., Kubernetes) by supporting a YAML [39]-compatible syntax, where users define scheduling policies in a terse, declarative way. To define function-specific policies, we assume to associate each function with a tag; then, in APP we name each policy with a tag, so that, at runtime, the Controller can pair each function with its APP policy and follow the latter's scheduling logic.[2]

We report the syntax of APP in Figure 2, as found in [20], of which we provide a brief overview in this section. In Section 3, we delve into more details with concrete examples.

The second assumption we make about the environment to run APP scripts is a 1-to-1 association so that each worker has a unique, identifying label. Indeed, the main, mandatory component of any policy (identified by a *policy_tag*) are the `workers` therein, i.e., a collection of labels that identify on which workers the scheduler can allocate the function. Specifically, each policy has a list of one or more *block*s, each including other two optional parameters besides the `worker` clause: the scheduling `strategy`, followed to select one of the workers of the block, and an `invalidate` condition, which determines when a worker cannot host a function. When a selected worker is invalid, the scheduler tries to allocate the function on the rest of the available workers in the block. If none of the workers of a block is available, then the next block is tried. The last clause, `followup`, encompasses a whole policy and defines what to do when no *block*s of the policy managed to allocate the function. When set to `fail`, the scheduling of the function fails; when set to `default`, the scheduling continues by following the (special) `default` policy.

We close by overviewing the options for both the `strategy` and `invalidate` parameters.

[2] The pairing of functions and policies is an orthogonal issue w.r.t. to APP. Indeed, one can obtain the same result with a 1-to-1 coupling between function identifiers and policies. However, we prefer the tag-based decoupling presented here because it is more flexible; e.g., it allows users to apply the same policy to multiple functions, as long as they are associated with the same tag.

$$
\begin{array}{lcl}
\mathit{policy\_tag} \in \mathit{Identifiers} \cup \{\texttt{default}\} & \mathit{worker\_label} \in \mathit{Identifiers} & n \in \mathbb{N} \\
\mathit{app} & ::= & \overline{\mathit{tag}} \\
\mathit{tag} & ::= & \mathit{policy\_tag}\ \texttt{:}\ \overline{\texttt{-}\ \mathit{block}}\ \mathit{followup}? \\
\mathit{block} & ::= & \texttt{workers:}\ [\ \texttt{*}\ |\ \overline{\texttt{-}\ \mathit{worker\_label}}\ ] \\
 & & (\texttt{strategy:}\ [\ \texttt{random}\ |\ \texttt{platform}\ |\ \texttt{best\_first}\ ])? \\
 & & (\texttt{invalidate:}\ [\ \texttt{capacity\_used}\ \texttt{:}\ n\texttt{\%} \\
 & & \qquad\qquad |\ \texttt{max\_concurrent\_invocations:}\ n \\
 & & \qquad\qquad |\ \texttt{overload}\ ])? \\
\mathit{followup} & ::= & \texttt{followup:}\ [\ \texttt{default}\ |\ \texttt{fail}\ ]
\end{array}
$$

Figure 2 The APP syntax.

The `strategy` parameter has 3 alternatives:

- `platform` indicates the usage of the platform-specific scheduling logic for the workers of the block. In practice, it delegates the scheduling decision to the platform's default scheduler (e.g., in OpenWhisk, it would use the original "co-prime" scheduling logic, cf. Section 2.1). This strategy is the default, when `strategy` is omitted, i.e., when there is no particular need to change the scheduler behaviour for a given function.
- `random` allocates functions stochastically among the workers of the block, following a uniform distribution. This strategy is useful in scenarios where balancing the distribution of functions over workers is more important than avoiding cold starts (e.g., when the system usually receives bursts of invocations).
- `best-first` allocates functions on workers based on their top-down order of appearance in the block. This strategy can decrease functions' run time, e.g., when workers have different performances and the user orders them by their best-to-worst performance ranking.

The `invalidate` parameter allows users to define when a worker becomes invalid to execute a given function. For instance, the hardcoded policy in OpenWhisk for worker invalidation is that it either exhausted its memory capacity (where each function consumes 256 MB) or reached the limit of 30 concurrent function invocations.

APP supports 3 `invalidate` options:

- `overload` (the standard one, when `invalidate` is omitted) captures the hardcoded invalidation strategy of the platform (e.g., in OpenWhisk, it translates to the exhaustion of the memory capacity or 30 concurrent invocations);
- `capacity_used`, associated with a percentage value, determines the threshold of memory capacity that declares a worker invalid;
- `max_concurrent_invocations`, associated with an integer value, sets the upper limit of concurrent invocations allowed on a worker.

Note that both `max_concurrent_invocations` and `capacity_used` are refinements of the `overload` option, thus their effect is to reduce respectively the maximum number of concurrent invocations or the memory capacity of the `overload` option.

3 APP Running Example

We show how we can use APP to control the scheduling of functions through an incremental, running example. We start with separate functions and their scheduling policy cases of increasing complexity and conclude by tying these functions together into a single FaaS application and related APP script.

3.1 Step 0: A Simple, Pure Function

The lowest level of complexity for our application is a single, lightweight, pure function, i.e., a function whose output depends only on its input, and requires no interaction with external services; examples of this category are functions performing mathematical operations, pre-processing of data, etc. Since we assume no interactions with external APIs or databases nor other requirements on workers that could impact the scheduling, we can fall back to the vanilla, hardcoded scheduling policy of the host serverless platform. When there is no need for specific APP-based policies, the standard `default` policy captures the basic behaviour of the underlying platform.

```
- default:
  - workers : *
    strategy: platform
    invalidate: overload
```

If no function has a specific tag, they fall under the `default` tag policy. This tag only has one block, which targets all workers (`*` match all the possible works labels) and adopts the `platform` `strategy` and the `overload` `invalidate` condition.

3.2 Step 1: Handling Locality when Querying External Databases

For the next step in complexity, let us assume we have some functions that use an external data source. Serverless functions indeed usually have limits on the size of their input payloads (e.g., AWS Lambda limits the payload size to 256KB and 6MB for asynchronous and synchronous requests respectively)[3] and they normally interact with data storage services, like databases, to retrieve (and store) the data they work on.

The addition of interaction with databases can cause latency problems due to data locality, i.e., the latency of the function is higher if database access is slow (e.g., due to long-distance interactions or narrow bandwidth). Scheduling a function's execution on a poorly-performing worker might incur latency overheads. Let us assume that the database is in Canada and that we have three workers in the region: `Canada_worker0`, `Canada_worker1`, `Canada_worker2`. Let us also assume that the query functions are tagged with the string `queries`. To schedule the functions only on those workers we can use the following script.

```
- queries:
  - workers:
    - Canada_worker0
    - Canada_worker1
    - Canada_worker2
    strategy: platform
  followup: default
```

[3] `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`.

We add the queries tag to label our new locality-bound functions, and require them to be scheduled only on workers that are located in Canada. We still keep the platform strategy, as we are not interested in giving any additional priority to one of the workers.

Note that we set the followup to default. This captures the fact that, while this kind of functions should be scheduled on workers that are close to the database (Canada), there is no strict functional requirement for it. Therefore, if neither of the preferred workers is available at the time of scheduling, it is acceptable for the function to be scheduled on some different workers, following the default policy.

3.3 Step 2: Prioritising Workers and Customising Invalidation

Building on the previous step, let us assume that the workers in Canada are not equivalent in terms of performance and we have a powerful worker labelled Canada_worker_Large, a less powerful worker labelled Canada_worker_Small, and a powerful worker with limited bandwidth labelled Canada_worker_Large_Narrow_Bandwidth.

We expect that the query function would perform the best on the Canada_worker_Large worker since it is both powerful and has wide bandwidth. When this worker is invalid, we have to choose between Canada_worker_Large_Narrow_Bandwidth and Canada_worker_Small. In this example, we prioritise the latter, as the computational limitations should come into play only when tasked with a relatively high number of invocations, while the former's reduced bandwidth might have a stronger effect on overall latency.

Moreover, while we want our workers to be prioritised according to their expected performance, we want to limit the resources that our functions are going to take to avoid scheduling too many functions in one worker, thus risking slowing down their run time. We set a limit of 75% memory load threshold for invalidation so that we never allocate more than 3/4 of a given worker's capacity.

To satisfy all new requirements, we update the configuration script as follows.

```
- queries:
  - workers:
    - Canada_worker_Large
    - Canada_worker_Small
    - Canada_worker_Large_Narrow_Bandwidth
    strategy: best_first
    invalidate:
      capacity_used: 75%
  followup: default
```

Here, we list sequentially the workers in order of priority, changing the strategy to best_first and setting the invalidate condition as discussed above.

3.3.1 Step 2.5: Invalidation Conditions and Block Lists

In the previous script, all three workers share the same invalidate condition, which can be undesirable when the difference in computational power among the nodes is sensible. If more fine-grained control is needed, we could define multiple blocks for the queries tag. Since APP tries to schedule functions within a given policy block by block, in their top-to-bottom order of appearance, we can define a scheduling logic analogous to the best_first strategy seen in the previous section by distributing the workers in different blocks, each with their specific invalidate options and exploiting the ordering of blocks to impose priority among them. Specifically, we set to 8 the threshold of maximal concurrent functions running on the Large workers, while we set it to 2 for the Small worker.

```
- queries:
  - workers:
    - Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - Canada_worker_Small
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
```

3.4 Step 3: Enforcing Configurations and Discarding Defaults

Let us now broaden the components of our serverless application by adding, alongside our initial data-dependent functions, some computationally heavier functions. More precisely, we want also to schedule functions that specifically require the presence of a GPU on the worker, e.g., needed to execute vector-based calculations in parallel.

In the previous steps, we allowed `queries` functions to execute on any worker different from the preferred ones if all the latter are invalid, thanks to the `default` `followup` option. Here, we assume that the GPU-dependent functions cannot be scheduled on non-GPU hardware, either because the function can not be compiled on non-GPU hardware or because running in other nodes leads to overloading the CPUs, thus causing failures due to timeouts[4].

We capture this behaviour as follows.

```
- GPU:
  - workers:
    - GPU_worker0
    - GPU_worker1
    strategy: platform
  followup: fail
```

Unlike the previous examples, we now use the `fail` option for the `followup` keyword. This tells the scheduler to terminate the invocation with an error if the required workers are not available, without falling back to the `default` tag.

3.5 Step 4: Putting it all Together

Let us conclude this section of examples by tying all cases seen before into a consistent scenario. Indeed, this is the final step in complexity for our example application, where we combine all the features of APP seen so far.

[4] Serverless frameworks usually impose a maximal run time for functions that span a few minutes, e.g., OpenWhisk default maximal duration is 60 seconds.

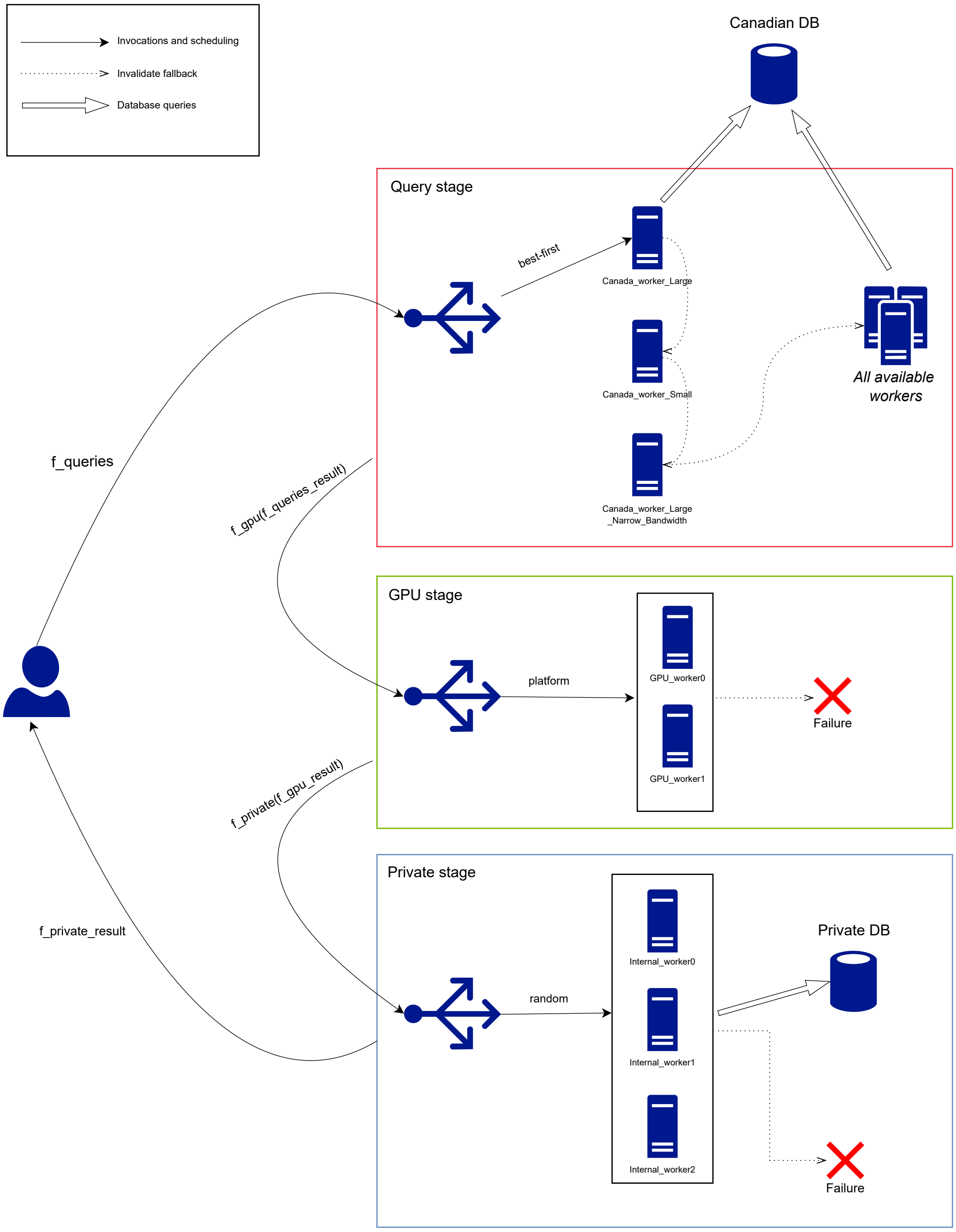


Figure 3 Architectural diagram of our example serverless application. Note how the `queries`-tagged functions fall back to select from all available workers when none of the preferred ones is available, by using the `default` policy.

Here, we assume that we are working in a hybrid cloud context, that is, we have both a private and a public part of our deployment: namely, besides the `Canada_worker`s and the `GPU_worker`s, previously presented and deployed on public cloud, we consider additional `Internal_worker`s deployed on-premises. Let us also assume that our private section has access to data inaccessible from the outside that can be used only by local workers, i.e. the additional `Internal_worker`s, for security reasons.

The application collects data from an external database, and subsequently pre-processes that data, using functions with the `queries` tag (discussed in Section 3.3.1). The output of these functions is then used to feed the `GPU` functions, which infer additional information from our data (discussed in Section 3.4). In the final step, the private database is queried and the received information is combined to enrich the information previously inferred. As such, we require the new functions implementing this final step to specifically target private workers.

To recapitulate, besides presenting the new `private` policy, we report the other policies seen in the previous sections, which make up the final APP script used to customise the scheduling of the functions in our example application.

```
- queries:
  - workers:
    - Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - Canada_worker_Small
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
- GPU:
  - workers:
    - GPU_worker0
    - GPU_worker1
    strategy: platform
  followup: fail
- private:
  - workers:
    - Internal_worker0
    - Internal_worker1
    - Internal_worker2
    strategy: random
    invalidate:
      capacity_used :80%
  followup: fail
- default:
  - workers : *
    strategy: platform
    invalidate: overload
```

The final architecture of the application, which we depict in Figure 3, consists of a pipeline with three stages. In the first stage, `queries`-tagged functions retrieve data from a Canadian database and perform some pre-processing over it, and then they invoke `GPU`-tagged functions for machine learning tasks (second stage). In the third stage, the completed `GPU` functions invoke `private`-tagged functions which combine the processed data with private data accessible only to the `Internal_workers` (see the `worker` clause under the `private` policy in the code above). Note that the `default` tag (at the end of the reported snippet) can be used by the `queries`-tagged functions, but not by the `GPU` and `private` functions, which all `fail` in case all their workers are invalid. When the `private`-tagged functions are invoked, only the private part of the system (the one containing the `Internal` workers and private database) is involved since the `private` tag lists only the `Internal` workers. In this example, we set the `strategy` to `random` to uniformly distribute the functions over the three workers. The `followup` set to `fail` naturally captures the fact that it would be pointless to run the `private` functions outside the private part of the system, since those workers would not be able to reach the private database and the `private` functions would fail.

4 APP Extensions

The original version of APP, presented in the previous sections, inspired extensions that introduced new language primitives. In this section, we discuss a couple of primitives respectively found in an extended version of APP published in [19] and from ongoing work, motivating them with concrete examples.

4.1 Targeting sets of workers

In cloud settings, the addition or removal of computing nodes is often done unbeknownst to the applications running on them. As a consequence, if the function scheduling language allows for referring to the computing nodes only individually, it may be impossible to define scheduling policies that exploit dynamic scenarios where nodes can change at runtime. An extension of APP to handle these more dynamic scenarios is to drop the 1-to-1 relation imposed between labels and workers. Specifically, we can allow the same worker to have multiple labels (e.g., a unique one to identify it and multiple, shared ones) and then add an APP primitive to express when we intend that a label induces a collection of workers.

As an example, we change the scenario in Section 3.3.1 to have three *families* of workers (instead of three workers): the `Large`, `Small`, and `Large_with_Narrow_Bandwidth` tiers.

In [19], we extended the APP language with the possibility to use expressions to match various tags of workers. Here, we present a refinement of that idea, introducing the `set` keyword to indicate that we intend worker labels as shared among a collection of workers, which the scheduler can choose among. As an example, the following APP script adds the `set` keyword before the workers' label found in the script from Section 3.3.1 to indicate that APP shall interpret it as the collection of workers associated with that label.

```
- queries:
  - workers:
    - set: Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - set: Canada_worker_Small
```

```
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - set: Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
```

Thus, the set keyword allows users to capture dynamic scenarios where workers change at runtime without requiring them to update the script when the underlying infrastructure change. Moreover, the new keyword allows us to handle large amounts of workers in a cleaner way, making scripts more easily writable and readable.

4.2 Function Anti-affinity

A second primitive, subject of ongoing work, is an anti-affinity option for the invalidate condition. This option allows policies to invalidate a worker when it hosts one or more functions that are deemed by the user anti-affine with the one under scheduling. Anti-affinity is important, e.g., for security reasons since there could be functional requirements that require avoiding running critical functions on a worker with other, unknown functions, which could exploit possible limitations of the runtime isolation to surreptitiously gather information from the former. Anti-affinity constraints may also play a role in the optimisation of the application performance. For example, let us consider a refinement of the scenario presented in Section 3.4 where the `GPU_Workers` have only 1 GPU each. In this context, it would be ideal to avoid scheduling another GPU-intensive function on the same worker, as this could impact the performance of both functions.

Adding an anti-affinity invalidate option allows us to capture this scenario with minimal modifications from the script presented in Section 3.4.

```
- GPU:
  - workers:
    - set: GPU_Worker
      invalidate:
        anti-affinity: GPU
  followup: fail
```

In the code above, the scheduler avoids placing `GPU` functions on a `GPU_Worker` if it hosts already a function associated with the same tag. In the example, in particular, we declared any function with tag `GPU` anti-affine with any other function with the same tag. This allows the execution of at most one function with tag `GPU` in all the workers with tag `GPU_Worker`.

We highlight that the introduction of anti-affinity constraints may be problematic for serverless applications of considerable size and introduce interesting research challenges. Indeed, to sustain high-traffic situations, serverless platforms (like OpenWhisk) consider the presence of multiple controllers, so that they can share the load of the inbound function invocations and avoid creating architectural bottlenecks. However, having multiple controllers can introduce scheduling races, so that multiple controllers asynchronously schedule functions on the same worker (this effect is generally not a problem since, at worst, the worker would non-deterministically reject allocations that exceed its capacity limits). In the context of anti-affinity, this issue becomes more relevant. Indeed, to guarantee the respect of anti-affinity constraints one might need to sensibly alter the serverless platform architecture. For example, one can use global locks – which might determine sensible performance degradation – or require the partitioning of workers among the available controllers – which would thwart the principle of resource sharing of cloud computing.

5 Discussion and Conclusion

We presented a tutorial on APP, an innovative approach to providing fine-grained control over serverless function scheduling to users. The interested reader can retrieve an OpenWhisk extension that supports APP-based scripts at [7].

To the best of our knowledge, APP is the first platform-agnostic configuration language for serverless scheduling. As serverless computing is gaining wider adoption [11, 27], many proposals tackled the problem of improving serverless function scheduling under different application contexts (and locality principles). In particular, there are several works focused on optimising serverless function scheduling focusing on improving the cold-start problem. These include techniques focused on container re-utilization and function scheduling heuristics and dedicated balancing algorithms [31, 27, 36, 35, 42, 1, 49, 3, 41]. Other research directions in the field regard the programming of compositions of serverless functions and the application of the Serverless paradigm to contexts such as Fog/Edge and IoT Computing. Examples of the first direction are proposals of calculi to formally reason on serverless functions and their implementations [22, 29] as well as proposals of the elements underpinning runtime support for compositions-as-functions [12]. The second direction sees studies on the emergence of real-time and data-intensive applications for edge computing and proposed a serverless platform designed for it, as well as frameworks for supporting cloud-to-edge serverless computing [14, 8, 25, 23].

Regarding APP's evolution, we want to explore the advantages offered by APP from both the language and implementation perspectives. On the latter, we plan to extend the number of platforms that support APP, besides our initial prototype based on OpenWhisk. As a prerequisite to support APP, we need nodes to be labelled. Looking at the architectures of alternative open-source serverless platforms, like OpenFaaS and Knative, adding support for one such prerequisite would follow the implementation we developed for OpenWhisk. Although we do not have precise knowledge of the internals of closed-sourced alternatives, e.g., AWS Lambda and Azure Functions, they likely do not label nodes or expose this information to the scheduler (since they do not support scheduling policies based on the identification of single nodes or groups thereof). However, since FaaS platforms tend to all share the same architecture, we deem it plausible to use our development on OpenWhisk as a guideline to add support for node labelling also in closed-source platforms.

From the language perspective, we propose to extend APP, focusing on the exploration of locality principles (e.g., code and session locality) and in providing users with constructs able to support custom definitions of `strategy` and `invalidate` options directly in the source APP configuration (also via shareable and importable modules). These extensions would enable greater flexibility and customisation of scheduling policies. We are also interested in studying heuristics that, based on the monitoring of existing serverless applications can suggest optimising scheduling policies. We deem configurator optimisers [2, 5] a good starting point for this activity, which can be extended to automatically generate policies based on developer's requirements. Finally, we propose to investigate the separation of concerns between developers and providers, to minimise the information that providers have to share to allow developers to schedule functions efficiently. This balancing would involve hiding the complexity of providers' dynamically changing infrastructure, while also allowing developers to customise their scheduling policies to meet their needs.

References

1 Cristina L. Abad, Edwin F. Boza, and Erwin Van Eyk. Package-aware scheduling of faas functions. In *Proc. of ACM/SPEC ICPE*, pages 101–106. ACM, 2018. doi:10.1145/3185768.3186294.

2 Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, pages 229–245, 2016. doi:10.1007/978-3-319-47677-3_15.

3 Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. of USENIX/ATC*, pages 923–935, 2018. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

4 Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018. doi:10.1145/3276488.

5 Amazon Web Services. AWS Compute Optimizer. https://aws.amazon.com/compute-optimizer/, aug 2023.

6 Apache kafka. https://kafka.apache.org/, aug 2023.

7 APP-based openwhisk extension. https://github.com/giusdp/openwhisk, aug 2023.

8 Austin Aske and Xinghui Zhao. Supporting multi-provider serverless computing on the edge. In *ICPP, Workshop Proceedings*, pages 20:1–20:6. ACM, 2018. doi:10.1145/3229710.3229742.

9 Aws lambda. https://aws.amazon.com/lambda/, aug 2023.

10 Microsoft azure functions. https://azure.microsoft.com/, aug 2023.

11 Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017. doi:10.1007/978-981-10-5026-8_1.

12 Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *ACM Onward! 2017*, pages 89–103, 2017. doi:10.1145/3133850.3133855.

13 Ali Banaei and Mohsen Sharifi. Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform. *The Journal of Supercomputing*, sep 2021. doi:10.1007/S11227-021-04057-Z.

14 Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE ICFC*, pages 1–10. IEEE, 2019. doi:10.1109/ICFC.2019.00008.

15 Luciano Baresi and Giovanni Quattrocchi. Paps: A serverless platform for edge computing infrastructures. *Frontiers in Sustainable Cities*, 3:690660, 2021.

16 Giuliano Casale, Matej Artač, W-J Van Den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020. doi:10.1007/S00450-019-00413-W.

17 IBM Cloud. Ibm cloud functions. https://cloud.ibm.com/functions/, aug 2023.

18 Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, pages 939–950, 2020. doi:10.1145/3366423.3380173.

19 Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. A declarative approach to topology-aware serverless function-execution scheduling.

In *2022 IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 11–15, 2022*. IEEE, 2022. doi:10.1109/ICWS55610.2022.00056.

20 Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. Allocation priority policies for serverless function-execution scheduling optimisation. In *Proc. of ICSOC*, volume 12571 of *LNCS*, pages 416–430. Springer, 2020. doi:10.1007/978-3-030-65310-1_29.

21 DigitalOcean. Digitalocean functions. https://www.digitalocean.com/products/functions/, aug 2023.

22 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less - A formal model for serverless computing. In *Proc. of COORDINATION*, volume 11533 of *LNCS*, pages 148–157. Springer, 2019. doi:10.1007/978-3-030-22397-7_9.

23 Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078468.3078497.

24 Google cloud functions. https://cloud.google.com/functions/, aug 2023.

25 Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 225–236, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302505.3310084.

26 Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29, 2021. doi:10.1186/S13677-021-00253-7.

27 Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

28 Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *Proc. of USENIX HotCloud*, 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

29 Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. of ACM on Prog. Lang.*, 3(OOPSLA):1–26, 2019. doi:10.1145/3360575.

30 Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proc. of ACM SIGOPS SOSP*, pages 691–707, New York, NY, USA, 2021. ACM. doi:10.1145/3477132.3483541.

31 Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019. URL: http://arxiv.org/abs/1902.03383.

32 Stefan Kehrer, Jochen Scheffold, and Wolfgang Blochinger. Serverless skeletons for elastic parallel processing. In *2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE*, pages 185–192, 2019.

33 Daniel Kelly, Frank Glavin, and Enda Barrett. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 304–312. IEEE, 2020. doi:10.1109/CLOUD49709.2020.00050.

34 Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *Proc. of USENIX ATC*, pages 805–820. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/kotni.

35 Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans-Arno Jacobsen. A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In *Proc. of Middleware (Posters)*, pages 3–4, 2018. doi:10.1145/3284014.3284016.

36 Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *Proc. of HotCloud 19*, Renton, WA, jul 2019. USENIX Association. URL: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

37 Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

38 Apache openwhisk. https://openwhisk.apache.org/, aug 2023.

39 Ingy döt Net Oren Ben-Kiki, Clark Evans. Yaml ain't markup language (yaml™) version 1.2. https://yaml.org/spec/1.2.2/, 2021.

40 APP-based openwhisk deployment. https://github.com/giusdp/ow-gcp, aug 2023.

41 Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *Middleware*, Middleware '17, pages 121–133. ACM, 2017. doi:10.1145/3135974.3135980.

42 Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proc. of MICRO*, pages 1063–1075, 2019. doi:10.1145/3352460.3358296.

43 Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of USENIX ATC*, pages 205–218, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

44 Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. of USENIX ATC*, pages 419–433. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

45 Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proc. of Middleware*, Middleware '20, pages 1–13, New York, NY, USA, 2020. ACM. doi:10.1145/3423211.3425682.

46 Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 17–25. IEEE, 2022. doi:10.1109/ICFEC54809.2022.00010.

47 Khondokar Solaiman and Muhammad Abdullah Adnan. Wlec: A not so cold architecture to mitigate cold start problem in serverless computing. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 144–153, 2020. doi:10.1109/IC2E48712.2020.00022.

48 Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020. URL: http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf.

49 Manuel Stein. The serverless scheduling problem and noah. *CoRR*, abs/1809.06100, 2018. arXiv:1809.06100.

50 Amoghavarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proc. of WOSC@Middleware*, pages 19–24. ACM, 2019. doi:10.1145/3366623.3368136.

51 Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

Model-Driven Code Generation for Microservices: Service Models

Saverio Giallorenzo
Università di Bologna, Italy
INRIA, Sophia Antopolis, France

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark

Marco Peressotti
University of Southern Denmark, Odense, Denmark

Florian Rademacher
Software Engineering, RWTH Aachen University, Germany

Abstract

We formally define and implement a translation of domain and service models expressed in the LEMMA modelling ecosystem for microservice architectures to source code in the Jolie microservice programming language. Specifically, our work extends previous efforts on the generation of Jolie code to the inclusion of the LEMMA service modelling layer.

We also contribute an implementation of our translation, given as an extension of the LEMMA2Jolie tool, which enables the practical application of our encoding. As a result, LEMMA2Jolie now supports a software development process whereby microservice architectures can first be designed by microservice developers in collaboration with domain experts in LEMMA, and then be automatically translated into Jolie APIs. Our tool can thus be used to enhance productivity and improve design adherence.

2012 ACM Subject Classification Software and its engineering → Software development methods; Applied computing → Service-oriented architectures; Software and its engineering → Model-driven software engineering

Keywords and phrases Microservices, Model-driven Engineering, Code Generation, Jolie APIs

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/jolie/lemma2jolie`
archived at `swh:1:dir:66cd1f01a83c6b26f7b22edc76291cf7ed7cc460`

Funding *Fabrizio Montesi*: Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

1 Introduction

Microservice Architecture (MSA) has risen to be a popular approach [24], but it also presents challenges related to design, development, and operation [5, 35]. To tackle design and development, researchers in software engineering and programming languages have proposed linguistic approaches to MSA, which feature high-level abstractions aimed at making microservice concerns more visible.

Model-Driven Engineering (MDE) [2] is a popular method for designing service architectures [1]. MDE can be applied to MSA by means of modelling languages such as MicroBuilder, MDSL, LEMMA, and JHipster [37, 39, 30, 15]. LEMMA, in particular, has been validated in real-world applications [36, 31]. On the side of development, Ballerina and Jolie [25, 22] are programming languages oriented towards services and their coordination. Jolie's abstractions have been found to improve productivity in industry [13], and LEMMA's support for Domain-Driven Design has been validated in real-world applications [36, 31].

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 6; pp. 6:1–6:17
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In recent work, Giallorenzo et al. [10] observed that the metamodels of LEMMA and Jolie have numerous contact points. This motivated the quest for integrating the two tools and their approaches, which in the long term could bring (quoting from [10])

> "*an ecosystem that coherently combines MDE and programming abstractions to offer a tower of abstractions [18] that supports a step-by-step refinement process from the abstract specification of a microservice architecture to its implementation*".

In other words, the objective is building a toolset that allows for (i) designing an MSA using the principles of MDE, and then (ii) seamlessly switching to implementing the design with a programming language that offers dedicated linguistic support for coding microservices. Achieving this objective requires integrating three elements of the metamodels of both LEMMA and Jolie [10]:

1. *Application Programming Interfaces* (API), describing what functionalities (and their data types) a microservice offers to its clients;
2. *Access Points*, capturing where and how clients can interact with a microservice's API;
3. *Behaviours*, defining the internal business logic of a microservice.

In [9], we started addressing the first element, by presenting an encoding, and a tool built on such encoding, that translates a large fragment of LEMMA's Domain Data Modelling Language (DDML) to Jolie types and interfaces. However, this encoding ignored the important aspects of modelling services, and in particular their interfaces in terms of operations and their associated communication patterns (e.g., synchronous vs asynchronous data provision). In this paper, we aim to bridge this gap and obtain the first prototype of an API generator from LEMMA service models.

Since the API is the layer the other two build upon, in this paper we focus on concretising the relationship between LEMMA and Jolie API layers. To this end, we extend previous work focused on a formal encoding from a large fragment of LEMMA's Domain Data Modelling Language (DDML) to Jolie types and interfaces [9]

Our key contribution is extending the encoding in [9] to a significant fragment of LEMMA's Service Modelling Language (SML); the one used for defining a set of microservices with their interfaces, operations, and accompanying communication patterns. Our extended encoding supports the systematic translation of LEMMA domain models – which, following Domain-Driven Design (DDD) [6] principles, capture domain-specific types including operation signatures – to Jolie APIs. As a second contribution, we extend the tool presented in [9], called LEMMA2Jolie, to accept both DDML and SML models and translate these into Jolie APIs, following the extended version of the encoding presented in this paper.

Taken together, these contributions constitute a new milestone on the roadmap traced in [10] for building a conceptual and technical bridge between the communities of programming languages and MDE on microservices. Specifically, our previous work made domain information from microservices' design actionable [9]. Here, we build upon our previous work [9] and move forward by adding support for the Service Viewpoint in MSA engineering [30]. While domain modelling is essential to most software systems and independent of the implemented architectural style, service modelling is essential to MSA, as it reifies the foundational concepts of information hiding and component interfacing. Therefore, this contribution completes previous work on APIs [9], and is pivotal for future activities that address the remaining elements, i.e., Access Points and Behaviours.

The remainder of the paper is organised as follows. Section 2 presents modelling concepts from LEMMA's DDML and SML and the relevant elements of the Jolie APIs required by the encoding, which we present in Section 3. Section 4 describes the implementation of LEMMA2Jolie and illustrates it with an example. Section 5 presents related work and a concluding discussion.

$$
\begin{array}{lcl}
CTX & ::= & \textbf{context}\ id\ \{\overline{CT}\} \\
CT & ::= & STR \mid COL \mid ENM \\
STR & ::= & \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\} \\
STRF & ::= & \textbf{aggregate} \mid \textbf{domainEvent} \mid \textbf{entity} \mid \textbf{factory} \\
 & \mid & \textbf{service} \mid \textbf{repository} \mid \textbf{specification} \mid \textbf{valueObject} \\
FLD & ::= & id\ id\ [\langle\overline{FLDF}\rangle] \mid S\ id\ [\langle\overline{FLDF}\rangle] \\
FLDF & ::= & \textbf{identifier} \mid \textbf{part} \\
OPS & ::= & \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \mid \textbf{function}\ (id \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \\
OPSF & ::= & \textbf{closure} \mid \textbf{identifier} \mid \textbf{sideEffectFree} \mid \textbf{validator} \\
COL & ::= & \textbf{collection}\ id\ \{(S \mid id)\} \\
ENM & ::= & \textbf{enum}\ id\ \{\overline{id}\} \\
S & ::= & \textbf{int} \mid \textbf{string} \mid \textbf{unspecified} \mid \ \ldots
\end{array}
$$

Figure 1 Simplified grammar of LEMMA's DDML [9]. Greyed out features are out of the scope of this paper and subject to future work.

2 Background

This section describes and exemplifies domain and service modelling with LEMMA, and the development of microservice APIs with Jolie.

2.1 LEMMA Domain Modelling Concepts

LEMMA's DDML supports domain experts and service developers in the construction of models that capture domain-specific types of microservices. We include the core grammar of this language in Figure 1 (grayed elements are not relevant for the translation presented in this work).[1]

LEMMA's DDML captures the foundational DDD concepts for MSA design. DDD's Bounded Context pattern [6] marks the boundaries of coherent domain concepts, thereby defining their scope and applicability [24]. A LEMMA domain model defines named bounded **context**s (rule CTX in Figure 1). A **context** may specify domain concepts in the form of complex types (CT), which are either structures (STR), collections (COL), or enumerations (ENM).

A **structure** gathers a set of data fields (FLD) each associated with a type that can be either a complex type from the same bounded context (id) or a built-in primitive type, e.g., **int** or **string** (S). LEMMA support continuous domain exploration by allowing the construction of underspecified models by means of the keyword **unspecified**. This concise solution provides domain experts and developers with a light-weight facility for refining models as they gain new domain knowledge [29]. **structure**s can comprise operation signatures (OPS) to reify domain-specific behaviour. An operation is either a **procedure** without a return type, or a **function** with a complex or primitive return type.

[1] The complete grammar can be found at `https://github.com/SeelabFhdo/lemma/blob/main/de.fhdo.lemma.data.datadsl/src/de/fhdo/lemma/data/DataDsl.xtext`.

$$
\begin{array}{lll}
SVR & ::= & \textbf{microservice}\ id\ \{\overline{IF}\} \\
IF & ::= & \textbf{interface}\ id\ \{\overline{IOP}\} \\
IOP & ::= & id\ (\overline{PAR}) \\
PAR & ::= & SYN\ DIR\ id : id \\
SYN & ::= & \textbf{sync} \mid \textbf{async} \\
DIR & ::= & \textbf{in}\ \mid \textbf{out}
\end{array}
$$

Figure 2 Simplified grammar of LEMMA's SML.

LEMMA's DDML supports the assignment of DDD patterns, called *features*, to structured domain concepts and their components. For instance, the **entity** feature (rule $STRF$ in Figure 1) expresses that a structure comprises a notion of domain-specific identity. The **identifier** feature then marks the data fields ($FLDF$) or operations ($OPSF$) of an **entity** which determine its identity.

The DDML also enables the modelling of **collection**s (rule COL in Figure 1), which represent sequences of primitives (S) or complex (id) values, as well as **enum**erations (ENM), which gather sets of predefined literals.

The following listing shows an example of a LEMMA domain model constructed with the grammar of the DDML [31].

```
context BookingManagement {
 structure ParkingSpaceBooking⟨entity⟩ {
  long bookingID⟨identifier⟩,
  double priceInEuro,
  function double priceInDollars
 }
}
                                                    LEMMA
```

The domain model defines the bounded **context** *BookingManagement* and its **structure**d domain concept *ParkingSpaceBooking*. It is a DDD **entity** whose *bookingID* field holds the **identifier** of an entity instance. The entity also clusters the field *priceInEuro* to store the price of a parking space booking, and the **function** signature *priceInDollars* for currency conversion of a booking's price.

2.2 LEMMA Service Modelling Concepts

We report in Figure 2 the (simplified) grammar of LEMMA's SML. Following the rules, we see that a LEMMA SML model can contain one or more **microservice**s, each associated with a name (id) and a collection of **interface**s. Each **interface** encloses a collection of operations, each identified by an id and a collection of PARameters. These parameters define the messaging pattern of the operation by associating each parameter – $id : id$, where the first id from the left is the name of the parameter and the second one is the name of its type (cf. Section 2.1) – with its timing of reception/transmission. Indeed, each parameter can either be **sync**hronous or **async**hronous and either be part of an **in**bound or an **out**bound message. We illustrate the matter with an example

$$
\begin{array}{lcl}
I & ::= & \textbf{interface}\ id\ \{[\textbf{RequestResponse}\ \overline{id(TP_1)(TP_2)}][\textbf{OneWay}\ \overline{id(TP)}]\} \\
TP & ::= & id \mid B \\
TD & ::= & \textbf{type}\ id :\ T \\
T & ::= & B\ [\{\overline{id\ C :\ T}\}] \mid \textbf{undefined} \\
C & ::= & [min, max] \mid * \mid \textbf{?} \\
B & ::= & \textbf{int}[(R)] \mid \textbf{string}[(R)] \mid \textbf{void} \mid \ldots \\
R & ::= & \textbf{range}([min, max]) \mid \textbf{length}([min, max]) \mid \textbf{enum}(...) \mid \ldots
\end{array}
$$

Figure 3 Simplified syntax of Jolie APIs (types and interfaces).

```
interface Sample {
    op(sync in a : int, async in b : int, sync out c : int, async out d : int)
}
                                                                        LEMMA
```

Above, we defined an **interface** called `Sample` which contains a single operation, *op*. The operation has four parameters. Starting from the leftmost, we find the parameter `a`, which is **sync**hronous and **in**bound. This means that `a` is part of the messages that *op* receives upon invocation. On the contrary, `b` is an **async**hronous **in**bound message, which means that it can reach *op* at any time between the invocation of *op* and its termination. Looking at the **out**bound parameters, we have `c` which is **sync**hronous, meaning that it is part of the message *op* sends when it terminated; `d`, on the contrary, is an **async**hronous **out**bound parameter, which *op* can transmit at any time between its invocation and its termination.

2.3 Jolie Types and Interfaces

Jolie interfaces and types define the functionalities of a microservice and the data types associated with those functionalities i.e., the API of a microservice. Figure 3 shows a simplified variant of the grammar of Jolie APIs, taken from [22] and updated to Jolie 1.10 (the latest major release at the time of writing).An **interface** is a collection of named operations (**RequestResponse**),where the sender delivers its message of type TP_1 and waits for the receiver to reply with a response of type TP_2 – although Jolie also supports **oneWay**s, where the sender delivers its message to the receiver, without waiting for the latter to process it (fire-and-forget), we omit them here because they are not used in the encoding (cf. Section 3). Operations have types describing the shape of the data structures they can exchange, which can either define custom, named types (id) or basic ones (B) (**int**egers, **string**s, etc.).

Jolie **type** definitions (TD) have a tree-shaped structure. At their root, we find a basic type (B) – which can include a refinement (R) to express constraints that further restrict the possible inhabitants of the type [7]. The possible branches of a **type** are a set of nodes, where each node associates a name (id) with an array with a range length (C) and a type T.

Jolie data types and interfaces are technology agnostic: they model Data Transfer Objects (DTOs) built on native types generally available in most architectures [4].

Based on the grammar in Figure 3, the following listing shows the Jolie equivalent of the example LEMMA domain model from Section 2.1.

```
///@beginCtx(BookingManagement)
///@entity
type ParkingSpaceBooking {
 ///@identifier
 bookingID: long
 priceInEuro: double
}
interface ParkingSpaceBooking_interface {
 RequestResponse:
  priceInDollars(ParkingSpaceBooking)(double)
}
///@endCtx
```
Jolie

Structured LEMMA domain concepts like *ParkingSpaceBooking* and their data fields, e.g., *bookingID*, are directly translatable to corresponding Jolie **type**s.

To map LEMMA DDD information to Jolie, we use Jolie documentation comments (*///*) together with an @-sign followed by the DDD feature name, e.g., *entity* or *identifier*. This approach enables to preserve semantic DDD information for which Jolie currently does not support native language constructs. The comments serve as documentation to the programmer who will implement the API. In the future, we plan on leveraging these special comments also in automatic tools (see Section 5).

LEMMA operation signatures are expressible as **RequestResponse** operations within a Jolie **interface** for the LEMMA domain concept that defines the signatures. For example, we mapped the domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface *ParkingSpaceBooking_interface* with the operation *priceInDollars*.

The following listing shows the Jolie equivalent of the example LEMMA service model from Section 2.2.

```
///@interface(Sample)
///@operationTypes(Sample.op)
type op_in {
 a : int
}
type op_out {
 c : int
}
type op_in_b {
 token:Token
 data : int
}
interface Sample {
 RequestResponse:
  op_in(op_in)(Token)
  op_out_d(Token)(int)
  op_out(Token)(op_out)
 OneWay:
  op_in_b(op_in_b)
}
```
Jolie

The operation *op* defined by the interface *Sample* contains asynchronous input and output parameters which do not have a direct equivalent in Jolie and thus need to be encoded. We propose to implement *op* into a series of request-response and one-way operations correlated

$$
\begin{aligned}
\llbracket \textbf{context}\ id\ \{\overline{CT}\} \rrbracket^{\mathrm{C}} &= \ ///\mathit{module}(id) \\
&\quad\ \overline{\llbracket CT \rrbracket^{\mathrm{C}}} \\
\llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\} \rrbracket^{\mathrm{O}} &= \ \overline{[///@STRF]}\ \textbf{interface}\ id\_interface\ \{\overline{\llbracket OPS \rrbracket^{\mathrm{O}}_{id}}\} \\
\llbracket \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \rrbracket^{\mathrm{O}}_{id_s} &= \ \textbf{RequestResponse}:\ \overline{[///@OPSF]}\ id(id\_type)(id_s) \\
\llbracket \textbf{function}\ (S \mid id_r)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \rrbracket^{\mathrm{O}}_{id_s} &= \ \textbf{RequestResponse}:\ \overline{[///@OPSF]}\ id(id\_type)((\llbracket S \rrbracket^{\mathrm{S}} \mid id_r)) \\
\llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\} \rrbracket^{\mathrm{C}} &= \ \ \textbf{type}\ \llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\} \rrbracket^{\mathrm{S}} \\
&\quad\ \ \overline{\llbracket OPS \rrbracket^{\mathrm{C}}_{id}}\ \llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{OPS}\} \rrbracket^{\mathrm{O}}_{id} \\
\llbracket \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \rrbracket^{\mathrm{C}}_{id_s} &= \ \textbf{type}\ id\_type:\ \textbf{void}\ \{\mathit{self}?:\ id_s\ \overline{\llbracket FLD \rrbracket^{\mathrm{S}}}\} \\
\llbracket \textbf{function}\ (id_r \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \rrbracket^{\mathrm{C}}_{id_s} &= \ \textbf{type}\ id\_type:\ \textbf{void}\ \{\mathit{self}?:\ id_s\ \overline{\llbracket FLD \rrbracket^{\mathrm{S}}}\} \\
\llbracket \textbf{collection}\ id\ \{(S \mid id_r)\} \rrbracket^{\mathrm{C}} &= \ \textbf{type}\ id:\ \textbf{void}\ \{\llbracket \textbf{collection}\ id\ \{(S \mid id_r)\} \rrbracket^{\mathrm{S}}\} \\
\llbracket \textbf{enum}\ id\ \{\overline{id}\} \rrbracket^{\mathrm{C}} &= \ \textbf{type}\ \llbracket \textbf{enum}\ id\ \{\overline{id}\} \rrbracket^{\mathrm{S}} \\
\llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\} \rrbracket^{\mathrm{S}} &= \ \overline{[///@STRF]}\ id:\ \textbf{void}\ \{\overline{\llbracket FLD \rrbracket^{\mathrm{S}}}\} \\
\llbracket S\ id\ [\langle\overline{FLDF}\rangle] \rrbracket^{\mathrm{S}} &= \ \overline{[///@FLDF]}\ id:\ \llbracket S \rrbracket^{\mathrm{S}} \\
\llbracket id_r\ id\ [\langle\overline{FLDF}\rangle] \rrbracket^{\mathrm{S}} &= \ \overline{[///@FLDF]}\ id:\ id_r \\
\llbracket \textbf{collection}\ id\ \{S\} \rrbracket^{\mathrm{S}} &= \ id*:\ \llbracket S \rrbracket^{\mathrm{S}} \\
\llbracket \textbf{collection}\ id\ \{id_r\} \rrbracket^{\mathrm{S}} &= \ id*:\ id_r \\
\llbracket \textbf{enum}\ id\ \{\overline{id}\} \rrbracket^{\mathrm{S}} &= \ id:\ \textbf{string}(\mathit{enum}(\overline{''id''})) \\
\llbracket \textbf{int} \rrbracket^{\mathrm{S}} &= \ \textbf{int} \\
\llbracket \textbf{unspecified} \rrbracket^{\mathrm{S}} &= \ \textbf{undefined}
\end{aligned}
$$

Figure 4 Salient parts of the Jolie encoding for LEMMA's domain modelling concepts [9].

by a *Token*. The first is the request-response *op_in* which takes the synchronous inputs of *op* and returns the token to be used to invoke the operation to provide and retrieve the remaining parameters of the operation. The asynchronous input b is provided to the implementation of *op* by means of the one-way operation *op_in_b* which takes as argument the token provided by *op_in* and the value for b. The asynchronous output d is retrieved by invoking *op_out_d* with the given token and the synchronous output c by invoking *op_out*. This encoding leverages Jolie's behavioural language which allows the definition of sophisticated interactions among a client and a service within the same session.

3 Encoding LEMMA Domain and Service Models as Jolie APIs

In this section we extend the encoding from LEMMA Domain Models to Jolie APIs presented in [9] (Section 3.1) to support also Service Models (Section 3.2).

3.1 Encoding LEMMA Domain Models [9]

We recap the description of the encoding from LEMMA domain models to Jolie from [9].

The encoding of LEMMA domain models is reported in Figure 4 and consists of three encoders: the *context* encoder $\llbracket \cdot \rrbracket^{\mathrm{C}}$ walks through the structure of LEMMA domain models to generate Jolie APIs using the encoders for *operations* ($\llbracket \cdot \rrbracket^{\mathrm{O}}$) and for *structures* ($\llbracket \cdot \rrbracket^{\mathrm{S}}$), respectively.

The operations encoder $[\![\cdot]\!]^{\mathrm{O}}$ generates Jolie interfaces based on **procedure**s and **function**s in the given models by translating structure-specific operations into Jolie operations. Because Jolie separates data from code that can operate on it (operations) the encoding needs to decouple **procedure**s and **function**s from their defining structures as illustrated in Section 2.3 by the mapping of the LEMMA domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface *ParkingSpaceBooking_interface* with the operation *priceInDollars*.

Given a structure X, we extend the signature of its **procedure**s with a parameter for representing the structure they act on and a return type X for the new state of the structure, essentially turning them into functions that transform the enclosing structure. For instance, we regard a procedure with signature $(Y \times \cdots \times Z)$ in X as a function with type $X \times Y \times \cdots \times Z \to X$. This approach is not new and can be found also in modern languages like Rust [17, 38] and Python [27]. The operation synthesised by the $[\![\cdot]\!]^{\mathrm{O}}$ encoder accepts the *id_type* generated by the $[\![\cdot]\!]^{\mathrm{C}}$ encoder that, in turn, has a *self* leaf carrying the enclosing data structure (id_s). The encoding of **function**s follows a similar path. Note that, when encoding *self* leaves, we do not impose the constraint of providing one such instance (represented by the **?** cardinality), but rather allow clients to provide it (and leave the check of its presence to the API implementer).

The main encoder $[\![\cdot]\!]^{\mathrm{C}}$ and the structure encoder $[\![\cdot]\!]^{\mathrm{S}}$ transform LEMMA types into Jolie types. **context**s translate into modules and, similarly to other DDD features, using pairs of ///@*beginCtx*(*context_name*) and ///@*endCtx* Joliedoc comment annotations. All the other constructs translate into **type**s and their subparts. When translating **procedure**s and **function**s, the two encoders follow the complementary scheme of $[\![\cdot]\!]^{\mathrm{O}}$ and synthesise the types for the generated operations. The other rules are straightforward.

3.2 Encoding LEMMA Service Models

The encoding of LEMMA service models is reported in Figure 5. The microservice interface encoder $(\!| \cdot |\!)^{\mathrm{MI}}$ translates the interfaces of a microservice into Jolie interfaces using the encoders $(\!| \cdot |\!)^{\mathrm{RR}}$ and $(\!| \cdot |\!)^{\mathrm{OW}}$ to translate its operations and $(\!| \cdot |\!)^{\mathrm{OT}}$ to generate the types required by them. The encoding assumes that each microservice works within a single context and fixes a type Token for data used to correlate invocations to Jolie operations that implement the same LEMMA operation (as discussed in Section 2.3), e.g., a UUID.

The type encoder $(\!| \cdot |\!)^{\mathrm{OT}}$ generates for each operation (i) a type collecting all its synchronous input parameters, (ii) a type for all its synchronous output parameters and (iii) at type for each of its asynchronous input parameters (to pair them with the token). Asynchronous output parameters do not require dedicated types.

The operation encoder $(\!| \cdot |\!)^{\mathrm{RR}}$ generates the request-response operations required to implement a LEMMA operation. If the LEMMA operation has only synchronous parameters, then it can be directly implemented as a single Jolie operation (similarly to procedure and functions of LEMMA's DDML). If an operation has asynchronous parameters, then it is encoded using multiple operations: (i) one to accept the synchronous inputs which is invoked first and provides the token used by the subsequent operations; (ii) one for retrieving each asynchronous output given a token; and (iii) one for awaiting the end of the implemented operation and retrieve all the synchronous outputs. Asynchronous inputs are provided using one-way operations generated by the encoder $(\!| \cdot |\!)^{\mathrm{OW}}$.

$$
\begin{aligned}
(\!|\textbf{microservice}\ ms\ \{\overline{IF}\}|\!)^{\mathrm{MI}} &= \overline{(\!|IF|\!)^{\mathrm{MI}}_{ms}}\\
(\!|\textbf{interface}\ \mathit{if}\ \{\overline{IOP}\}|\!)^{\mathrm{MI}}_{ms} &= ///@\mathit{interface}(ms.\mathit{if})\\
&\quad \overline{(\!|IOP|\!)^{\mathrm{OT}}_{\mathit{if}}}\\
&\quad \textbf{interface}\ \mathit{if}\ \{\\
&\qquad \textbf{RequestResponse}:\ \overline{(\!|IOP|\!)^{\mathrm{RR}}_{\mathit{if}}}\\
&\qquad \textbf{OneWay}:\ \overline{(\!|IOP|\!)^{\mathrm{OW}}_{\mathit{if}}}\\
&\quad \}\\
\left(\!\left| op \begin{pmatrix} \overline{\textbf{sync in}\ id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in}\ id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out}\ id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out}\ id_{SO} : id'_{SO}} \end{pmatrix} \right|\!\right)^{\mathrm{OT}}_{\mathit{if}} &= \begin{aligned}[t] &///@\mathit{operationTypes}(\mathit{if}.op)\\ &\textbf{type}\ op\_in\{\overline{id_{SI} : id'_{SI}}\}\\ &\textbf{type}\ op\_out\{\overline{id_{SO} : id'_{SO}}\}\\ &\overline{\textbf{type}\ op\_in\_id_{AI}\{\mathrm{token} : \mathrm{Token}, \mathrm{data} : id'_{AI}\}} \end{aligned}\\
\left(\!\left| op \begin{pmatrix} \overline{\textbf{sync in}\ id_{SI} : id'_{SI}}, \\ \overline{\textbf{sync out}\ id_{SO} : id'_{SO}} \end{pmatrix} \right|\!\right)^{\mathrm{RR}}_{\mathit{if}} &= \begin{aligned}[t] &///@\mathit{operation}(\mathit{if}.op)\\ &op(op\_in)(op\_out) \end{aligned}\\
\left(\!\left| op \begin{pmatrix} \overline{\textbf{sync in}\ id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in}\ id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out}\ id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out}\ id_{SO} : id'_{SO}} \end{pmatrix} \right|\!\right)^{\mathrm{RR}}_{\mathit{if}} &= \begin{aligned}[t] &///@\mathit{operation}(\mathit{if}.op)\\ &op\_in(op\_in)(\mathrm{Token})\\ &\overline{op\_out\_id_{AO}(\mathrm{Token})(id_{AO})}\\ &op\_out(\mathrm{Token})(op\_out) \end{aligned}\\
\left(\!\left| op \begin{pmatrix} \overline{\textbf{sync in}\ id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in}\ id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out}\ id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out}\ id_{SO} : id'_{SO}} \end{pmatrix} \right|\!\right)^{\mathrm{OW}}_{\mathit{if}} &= \begin{aligned}[t] &///@\mathit{operation}(\mathit{if}.op)\\ &\overline{op\_out\_id_{AI}(op\_in\_id_{AI})} \end{aligned}
\end{aligned}
$$

Figure 5 Jolie encoding for LEMMA's service modelling concepts.

4 LEMMA2Jolie and Example

4.1 LEMMA2Jolie

We implement our extended encoding (cf. Section 3) by including the parsing of SML models and the new rules of the encoding presented here into LEMMA2Jolie. LEMMA2Jolie is a tool that transforms LEMMA models into Jolie code and that was initially presented in [9] where we have shown the feasibility of producing Jolie code from LEMMA domain models. The additions to LEMMA2Jolie described in this paper target LEMMA service models and are relatively straightforward. Specifically, we integrate the parsing of the SML models, which generate an in-memory object graph containing types and service information. Then, these run through an execution engine for templates, that transforms the in-memory representation into Jolie code that the tool outputs in file format. We provide the extended version of LEMMA2Jolie in a permanent repository on Software Heritage[2].

[2] https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/jolie/lemma2jolie

4.2 Example

We exemplify the encoding of LEMMA service and domain models in Jolie APIs based on the Food to Go (FTGO) case study by Richardson [33]. FTGO consists of six microservices that realise the backend of a web application for online food ordering from local restaurants. The microservices are responsible for accounting, consumer handling, delivery management, kitchen management, order handling, and restaurant organization. In the following, we focus on FTGO's Order microservice which is responsible for handling food orders. The following listing shows an excerpt of the LEMMA domain model for the Order microservice[3].

```
context API {
 structure CreateOrderRequest⟨valueObject⟩ {
  immutable long consumerId,
  immutable long restaurantId,
  immutable LineItems lineItems
 }

 structure LineItem {
  string menuItemId,
  int quantity
 }

 collection LineItems { LineItem i }

 structure CreateOrderResponse⟨valueObject⟩ {
  immutable long orderId
 }

 structure GetOrderResponse⟨valueObject⟩ {
  immutable long orderId,
  immutable string state,
  immutable double orderTotal
 }
}
                                                              LEMMA
```

The domain model defines the *API* bounded context. It comprises five domain concepts:

- *CreateOrderRequest:* This domain concept is a DDD **valueObject** and as such responsible for encapsulating data that is shared between software components [6]. The data fields of value objects are usually **immutable** because they receive a value exactly once for data transmission. The Order microservice enables clients to communicate information relevant to food order placing using the *CreateOrderRequest* concept.
- *LineItem:* This domain concept models a single line item of some food order. Therefore, it identified the item on the available menu and its ordered quantity.
- *LineItems:* The *LineItems* concept gathers all line items of a food order. *CreateOrderRequest* concept relies on it to communicate a consumer's order to the selected restaurant.
- *CreateOrderResponse:* The Order microservice replies to *CreateOrderRequesta* with this **valueObject**. It clusters the identifier of the created order.

[3] The complete model can be found at `https://archive.softwareheritage.org/browse/revision/d4447fe8bfcaa319e540ed89d160d8fe817e128f/?origin_url=https://github.com/jolie/lemma2jolie&path=sample-2.data&revision=d4447fe8bfcaa319e540ed89d160d8fe817e128f`

- *GetOrderResponse:* Using this domain concept, the Order microservice provides clients with information about the state of a certain order, e.g., “accepted‘” or “cancelled”, and its total costs.

The following listing shows an example LEMMA service model for the Order microservice[4].

```
import datatypes from "sample-2.data" as Domain

functional microservice org.example.OrderService {
 interface Orders {
  createOrder(
   sync in request : Domain::API.CreateOrderRequest,
   sync out response : Domain::API.CreateOrderResponse
  );

  getOrder(
   sync in orderId : long,
   sync out response : Domain::API.GetOrderResponse
  );

  monitorOrder(
   sync in orderId : long,
   async out response : Domain::API.GetOrderResponse
  );
 }
}
                                                          LEMMA
```

The model **import**s the above domain model including the *API* bounded context. LEMMA’s import mechanism allows the composition of models for different viewpoints on a microservice architecture by enabling inter-model references [30]. The purpose of these references depends on the composed model kinds. For a service model that imports a domain model as shown in the listing, inter-model references support typing of microservice operation parameters with modelled domain concepts (see below). Import statements in LEMMA start with the **import** keyword followed by a keyword that identifies the kinds of imported elements, e.g., **datatypes** for domain concepts that are to be used as types for microservice operation parameters. After the **from** keyword, modellers specify the path to the imported model, i.e., “sample-2.data” in the listing[5], and a shorthand alias after the **as** keyword. Thus, in the service model, modellers can refer to the elements of the above domain model located in the file “sample-2.data” by the alias *Domain*.

After the **import** statement, the service model defines the **functional microservice** *org.example.OrderService*. In LEMMA’s SML, microservices must have at least one qualifying naming level like “org.example” to allow the semantic clustering of services [30]). The *OrderService* consists of a single **interface** called *Orders* that gathers the following operations:

---

4 The actual model can be found at `https://archive.softwareheritage.org/browse/content/sha1_git:267211533f271c8140166b3acc3729906baf3126/?origin_url=https://github.com/SeelabFhdo/lemma&path=examples/food-to-go/Order/Order.services`

5 Please note that the file “sample-2.data” comprises the previous domain model and that the filename indicates the fact that the file clusters the LEMMA model code for the second usage example of LEMMA2Jolie in its repository at `https://archive.softwareheritage.org/browse/revision/d4447fe8bfcaa319e540ed89d160d8fe817e128f/?origin_url=https://github.com/jolie/lemma2jolie`.

- *createOrder:* This operation supports the creation of food orders in FTGO. To this end, it expects the **sync**hronously **in**coming parameter *request* whose type is the domain concept *CreateOrderRequest* imported from the *API* bounded context of the above domain model. The **sync**hronously **out**going parameter *response* then represents the result of food order creation as reified by the imported concept *CreateOrderResponse*.
- *getOrder:* This operation enables the retrieval of information about placed food orders. Therefore, it requires the identifier of an order and informs callers about its state by means of the *GetOrderResponse* concept from the imported domain model. As for *createOrder*, *getOrder* interacts with clients in a fully synchronous fashion.
- *monitorOrder:* Similar to *getOrder*, this operation provides callers with information about food orders. However, it expects continuous querying for this information leveraging the **async**hronously **out**going parameter *response*. Thus, the operation can be used, e.g., by mobile apps to display notifications about order state changes without the need for re-establishing synchronous HTTP connections[6].

Based on our encoding (Sect. 3), LEMMA2Jolie produces the following Jolie code from the LEMMA service model and the imported domain model.

```
///@beginCtx(API)
///@valueObject
type CreateOrderRequest {
 consumerId: long
 restaurantId: long
 lineItems: LineItems
}
type LineItem {
 menuItemId: string
 quantity: int
}
type LineItems {
 i*: LineItem
}
///@valueObject
type CreateOrderResponse {
 orderId: long
}
///@valueObject
type GetOrderResponse {
 orderId: long
 state: string
 orderTotal: double
}
///@endCtx
                                                                    Jolie
```

The code between the Joliedoc comments *///@beginCtx(API)* and *///@endCtx(API)* represents the result of our encoding for LEMMA's domain modelling concepts (Sect. 2). The code following the Joliedoc comment *///@interface(org.example.OrderService.Orders)*, on the other hand, adheres to the novel encoding for LEMMA's service modelling concepts (Sect. 3).

[6] Note that *monitorOrder* is not part of the Order microservice's original interface [33]. Instead, we included this operation for illustration purposes.

That is, all synchronously typed parameters of the *createOrder*, *getOrder*, and *monitorOrder* receive a dedicated type per direction (*createOrder_in* and *createOrder_out*, *getOrder_in* and *getOrder_out*, and *monitorOrder_in*) given LEMMA's semantics of communicating synchronous data in coherent data transfer objects [4]. By contrast, each asynchronously typed parameter (*response* of *monitorOrder*) is mapped to a dedicated type (*monitorOrder_out_response*) to enable clients the sending and receipt of asynchronous data at arbitrary and decoupled points in operations' runtime.

```
///@interface(org.example.OrderService.Orders)
///@operationTypes(org.example.OrderService.Orders.createOrder)
type createOrder_in {
 request : CreateOrderRequest
}
type createOrder_out {
 response : CreateOrderResponse
}
///@operationTypes(org.example.OrderService.Orders.getOrder)
type getOrder_in {
 orderId : long
}
type getOrder_out {
 response : GetOrderResponse
}
///@operationTypes(org.example.OrderService.Orders.monitorOrder)
type monitorOrder_in {
 orderId : long
}
type monitorOrder_out_response {
 response : GetOrderResponse
}
interface org_example_OrderService_Orders {
 RequestResponse:
  createOrder(createOrder_in)(createOrder_out),
  getOrder(getOrder_in)(getOrder_out),
  monitorOrder_in(monitorOrder_in)(Token),
  monitorOrder_out_response(Token)(monitorOrder_out_response)
}
                                                        Jolie
```

As described in Sect. 3, interfaces of modelled LEMMA microservices are encoded as Jolie **interface**s. That is, from the *OrderService*'s *Orders* interface, LEMMA2Jolie produces the Jolie interface *org_example_OrderService_Orders* with four **RequestResponse** operations. *createOrder* and *getOrder* map to the eponymous, fully synchronous operations in the LEMMA service model for the *OrderService*. On the other hand, *monitorOrder_in* and monitorOrder_out_response reify different parts of the modelled *monitorOrder* operation. *monitorOrder_in* is the synchronous trigger for the execution of the *monitorOrder* logic. The result of the trigger's invocation is a *Token* that identifies the execution of the triggered *monitorOrder* instance. monitorOrder_out_response then requires the *Token* to provide clients with the instance's data that was modelled by the asynchronous *response* parameter.

5 Discussion, Related Work, and Conclusion

The use of MDE in both industrial and academic contexts, along with its effective support for developing intricate software systems, has led to the creation of numerous tools similar to LEMMA2Jolie [34, 16, 39, 37, 15]. These tools act as code generators within the conceptual framework of MDE [2] and generate artefacts for the engineering of MSA. They accomplish this task through models built using specific modelling languages.

Compared to LEMMA2Jolie, most of the related alternatives focus on Java as the target technology [34, 37, 15], rather than service-oriented programming languages. Contrarily, LEMMA2Jolie focuses on Jolie, which has been introduced to reduce the semantic gap between microservice concepts and implementation languages. Jolie's APIs are by design technology-agnostic and support their implementation with different transport protocols and technologies (e.g., Jolie, Java, JavaScript) [22, 20, 19]. Additionally, the modelling languages supported by the mentioned proposals and the resulting generated code only address single concerns in MSA engineering, such as domain modelling [34, 16] or service API implementation and provisioning [39, 37, 15]. In contrast, LEMMA's modelling languages provide an integrated solution for multi-concern modelling in MSA engineering by offering modelling languages for various microservice architecture viewpoints [30].

As described in Section 3 and Section 4, the encoding we specify and its implementation demonstrate the practicality of combining the LEMMA and Jolie ecosystems. There are several areas for future exploration, including extending the findings to other programming languages, examining the maturity of LEMMA2Jolie, formally proving the correctness of the encoding, and expanding the integration in different directions.

Interesting future work includes assessing the practical usefulness of LEMMA2Jolie. We mention a few possibilities, inspired also by best practices found in previous research on modelling languages [36, 31]. The first is to conduct controlled user experiments with practitioners, for example in order to evaluate how LEMMA2Jolie contributes to improving quality and productivity. Second, we could recruit practitioners to use LEMMA2Jolie, in order to evaluate their experience with using it and the result of their efforts. Finally, we could use LEMMA2Jolie to recreate existing microservice architectures written in Jolie and then compare the existing and obtained codebases in qualitative and quantitative terms [13, 11, 3]. Some of these architectures [11, 3] follow the API patterns recently identified in [39], and checking whether these patterns can be faithfully captured in LEMMA2Jolie could extend our knowledge on the connection between API patterns and MDE for microservices [31, 32].

To provide correctness guarantees of the encoding, we must first establish a formalisation of the semantics of both LEMMA's DDML and SML and Jolie APIs, and then prove that the encoding generates Jolie APIs that maintain the semantics of the input DDML and SML models. This effort is currently underway, as portions of Jolie have already been formalised [12, 21, 8, 22], and LEMMA implements context conditions [14] to restrict the proper formation of DDML models concerning their intended semantics [30].

We also intend to expand LEMMA2Jolie with capabilities for round-trip engineering (RTE). RTE accounts for the bidirectional synchronisation between models and generated code [26]. In the context of LEMMA2Jolie, RTE would further strengthen the collaboration between domain experts, who capture relevant application concepts in non-technical DDML models, and microservice developers, who adapt generated Jolie code to their needs and leverage RTE to reflect these changes back to the model-level for an efficient communication with domain experts.

The extension of LEMMA2Jolie presented in this paper forms the basis for future support of Access Point and Behaviour derivation from LEMMA models (Section 1). To this end, LEMMA2Jolie would have to consider further languages of LEMMA, e.g., the Technology Modeling Language [28] and Operation Modeling Language [30], in the translation towards Jolie code. Further along this direction, we plan to investigate the integration with [23] to automatically decompose the Jolie codebase generated by LEMMA2Jolie and synthesise suitable cloud deployment configurations.

References

1 David Ameller, Xavier Burgués, Oriol Collell, Dolors Costal, Xavier Franch, and Mike P. Papazoglou. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, 62:42–66, 2015. Elsevier. `doi:10.1016/J.INFSOF.2015.02.006`.

2 Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, 2017.

3 Luís Cruz-Filipe, Sofia Kostopoulou, Fabrizio Montesi, and Jonas Vistrup. μxl: Explainable lead generation with microservices and hypothetical answers. In Florian Rademacher and Jacopo Soldani, editors, *Service-Oriented and Cloud Computing - 10th IFIP WG 6.12 European Conference, ESOCC 2023, Larnaca, Cyprus, October 24-25, 2023, Proceedings*, Lecture Notes in Computer Science. Springer, 2023. `doi:10.1007/978-3-031-46235-1_1`.

4 Robert Daigneau. *Service Design Patterns*. Addison-Wesley, 2012.

5 Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017. `doi:10.1007/978-3-319-67425-4_12`.

6 Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

7 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of the 1991 Conf. on Programming Language Design and Implementation*, pages 268–277, 1991. `doi:10.1145/113445.113468`.

8 Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 21–40. Springer, 2018. `doi:10.1007/978-3-319-92612-4_2`.

9 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, and Florian Rademacher. Model-driven generation of microservice interfaces: From LEMMA domain models to jolie apis. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13271 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2022. `doi:10.1007/978-3-031-08143-9_13`.

10 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh. Jolie and LEMMA: Model-driven engineering and programming languages meet on microservices. In *Coordination Models and Languages*, pages 276–284. Springer, 2021. `doi:10.1007/978-3-030-78142-2_17`.

11 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Narongrit Unwerawattana. Jot: A jolie framework for testing microservices. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages*, volume 13908 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2023. `doi:10.1007/978-3-031-35361-1_10`.

12 Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: a calculus for service oriented computing. In *International Conference on Service-Oriented Computing*, pages 327–338. Springer, 2006. `doi:10.1007/11948148_27`.

13 Claudio Guidi and Balint Maschio. A jolie based platform for speeding-up the digitalization of system integration processes, 2019. URL: https://www.conf-micro.services/2019/papers/Microservices_2019_paper_6.pdf.

14 David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, oct 2004. IEEE. doi:10.1109/MC.2004.172.

15 JHipster. Jhipster domain language (jdl), 2022-14-02. URL: https://www.jhipster.tech/jdl.

16 Stefan Kapferer and Olaf Zimmermann. Domain-specific language and tools for strategic Domain-driven Design, context mapping and bounded context modeling. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, pages 299–306. INSTICC, SciTePress, 2020. doi:10.5220/0008910502990306.

17 Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

18 Robin Milner. The tower of informatic models. *From semantics to Computer Science*, 2009.

19 Fabrizio Montesi. Jolie: a Service-oriented Programming Language. Master's thesis, University of Bologna, Department of Computer Science, 2010. URL: http://amslaurea.cib.unibo.it/1226/.

20 Fabrizio Montesi. Process-aware web programming with Jolie. *Sci. Comput. Program.*, 130:69–96, 2016. doi:10.1016/J.SCICO.2016.05.002.

21 Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari Nezhad, editors, *Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings*, volume 7084 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2011. doi:10.1007/978-3-642-25535-9_9.

22 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014. doi:10.1007/978-1-4614-7518-7_4.

23 Fabrizio Montesi, Marco Peressotti, and Valentino Picotti. Sliceable monolith: Monolith first, microservices later. In Barbara Carminati, Carl K. Chang, Ernesto Daminai, Shuigung Deng, Wei Tan, Zhongjie Wang, Robert Ward, and Jia Zhang, editors, *IEEE International Conference on Services Computing, SCC 2021, Chicago, IL, USA, September 5-10, 2021*, pages 364–366. IEEE, 2021. doi:10.1109/SCC53864.2021.00050.

24 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015. URL: https://www.worldcat.org/oclc/904463848.

25 Andy Oram. *Ballerina: A Language for Network-Distributed Applications*. O'Reilly, 2019.

26 Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016. doi:10.1016/J.JSS.2015.08.047.

27 Python Software Foundation. The python language reference, 2021.

28 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Aspect-oriented modeling of technology heterogeneity in Microservice Architecture. In *2019 IEEE Int. Conf. on Software Architecture (ICSA)*, pages 21–30. IEEE, 2019. doi:10.1109/ICSA.2019.00011.

29 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 229–236. IEEE, 2020. doi:10.1109/SEAA51224.2020.00047.

30 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, Sabine Sachweh, and Albert Zündorf. Graphical and textual model-driven microservice development. In *Microservices: Science and Engineering*, pages 147–179. Springer, 2020. doi:10.1007/978-3-030-31646-4_7.

31 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards holistic modeling of microservice architectures using LEMMA. In *Companion Proc. of the 15th Europ.*

Conf. on Software Architecture. CEUR-WS, 2021. URL: `https://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf`.

32 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards an extensible approach for generative microservice development and deployment using lemma. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, *Software Architecture*, pages 257–280, Cham, 2022. Springer International Publishing. `doi:10.1007/978-3-031-15116-3_12`.

33 Chris Richardson. *Microservices Patterns*. Manning Publications, first edition, 2019.

34 Sculptor Team. Sculptor–generating Java code from DDD-inspired textual DSL, 2022-14-02. URL: `https://www.sculptorgenerator.org`.

35 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*, 146:215–232, 2018. Elsevier. `doi:10.1016/J.JSS.2018.09.082`.

36 Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):459, 2021. `doi:10.1007/S42979-021-00825-Z`.

37 Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018. Taylor & Francis. `doi:10.1080/17517575.2018.1460766`.

38 The Rust Foundation. The rust reference, 2021.

39 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, Boston, 2023.

Towards Self-Architecting Autonomic Microservices

Claudio Guidi
italianaSoftware s.r.l, Imola, Italy

Abstract

Autonomic computing is a key challenge for system engineers. It promises to address issues related to system configuration and maintenance by leaving the responsibility of configuration and reparation to the components themselves. If considered in the area of microservices, it could help in fully decoupling executing platforms from microservices because they permit to avoid coupling at the level of non functional features. In this paper, I explore the case of self-architecting autonomic microservices through the illustration of a proof of concept. The key points and the main challenges of such an approach are discussed.

2012 ACM Subject Classification Computer systems organization → Self-organizing autonomic computing

Keywords and phrases Autonomic computing, microservices, architectures

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.7

Supplementary Material *Software*: `https://github.com/klag/autonomic-microservices` [17]
archived at `swh:1:dir:8ddd029c52e9caa1743c0394a326b95c8623c422`

1 Introduction

In recent years, the push towards the digitisation of processes has led to an increase of system complexity, both in terms of the number of applications and integration processes. Such a fact had impacts both on the organizations and the system architectures.

As far as organizations are concerned, the necessity for software that is increasingly aligned with business needs, has led to the adoption of organizational processes targeted to minimize the delivering time and to increase the frequency of releases. The most important example in this case is represented by DevOps[12, 16] that is a software development approach used for reducing the distance between development and operational activities. On the other hand, if we consider the evolution of architectures, we have observed the raise of *microservices*, that is a distributed oriented architectural approach which introduces a new transformative force in the design and deployment phases of a software system. Following a microservices approach, functionalities are isolated by responsibility, independently deployed and distributed into the system in order to allow for independent scaling and management. Each microservice is designed, developed and managed by a different team where all the required competences are present. Both DevOps and microservices can be considered as two complementary forces that, when combined, aims to:(i) increase the organization's speed; (ii) create a software application, or more generally, a software system, by integrating multiple independent components. They contribute to make the final system more resilient, flexible and scalable. The price to pay is a general rise of the overall complexity of the systems. DevOps and microservice platforms, even if they are commercial or custom, play a fundamental role for addressing such a complexity. In real cases, these platforms are usually a mix of technologies that address different functionalities. As an example Jenkins[6] is used for programming automatic tasks for DevOps, GitLab[3] is used as a code repository, Docker[2] and Kubernetes[9] are used for managing the containerization layer, OpenShift[11] is used for addressing the infrastructural layer and so on.

Automation is a key aspect of DevOps, especially when applied in the context of microservices, as these considerably increase the number of deployed components and, therefore, increasingly require automated tools for their management. In this context, autonomic

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 7; pp. 7:1–7:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

computing is a key challenge for system engineers [21] and it may be considered as a further step forward in automating systems. It promises to address issues related to configuration, maintenance, updating and security by leaving all the responsibilities to the software itself without any human intervention. In the vision of autonomic computing, an autonomic component possesses all the capabilities for self-detecting errors, performance deterioration and security threats, and consequently take actions for repairing and adjusting their status. Moreover, they are able to detect when their internal modules need to be updated, and they are able to correctly install and configure the new versions of them. Differently from DevOps, that is developer and IT operator-oriented focusing on collaboration and automation in the software development lifecycle, autonomic computing aims to create fully autonomous and self-sufficient computing systems that can adapt, optimize, and protect themselves. While DevOps is a widespread system of practices applied in many different contexts, autonomic computing is still a new and little explored field, especially in production environments.

In this scenario, autonomic computing could be considered as a contribution to the evolution of DevOps, applied specifically to microservices, for reducing human intervention. In order to understand the role it could play, we can start by noting that, even if microservices should be agnostic with respect to the technologies used for developing, they are actually coupled with the platforms due to non functional constraints and limitations inherently present. For example the set of programming technologies could be limited because only some of them are managed in the existing DevOps pipelines. Indeed, developing and maintaining a DevOps pipeline for a given technology comes with an organizational cost that could be not convenient if the related stream of work is not relevant. Furthermore, since the observability of components and their fault management processes are often centralised and delivered exploiting different platforms, depending on how an organization approaches their management (e.g. following ITIL[5] strategies *Service Operation* and *Continual Service Improvement*), microservices must be equipped with specific connectors or even specially programmed to adhere to general guidelines of the organization. Summarizing, if from a functional point of view a microservice can be designed to be independent and decoupled with respect of the rest of the system, from a non functional point of view it could be strongly coupled with the platform where it is developed and deployed. Such a coupling could represent an issue when some modifications at the level of the platform must be performed. Depending on their impact, the risk is that all the microservices must be revised in order to adhere to the new platform standards. Moreover, in case of migration from a platform to another, an important refactor of the microservices must be considered and made. Minimising non-functional coupling between microservices and the platforms on which they are deployed can enable their truly independent design, development and deployment. Such a milestone could be achieved by introducing autonomic computing at the level of the microservices, thus making them independent and autonomous in managing non-functional properties w.r.t. the execution environment where they are deployed. At the present, in the current practices, microservices are not equipped with any self-adaptation logic, they are never aware of the context where they are executed. Every operational activity on a microservice, also those that are automatic and related to some non functional aspects like auto-scaling, are always demanded to the external platforms where they are executed.

In this paper, I investigate the possibility to make a step forward in the direction of a non functional decoupling between microservices and platforms by exploring the idea of self-architecting autonomic behaviours in microservices. In particular, I propose a proof of concept where an autonomic microservice is able to negotiate the scaling, and the de-scaling, of one of its internal components with the execution environment. The main contribution of this paper is to show how a microservice of this kind could be developed, which are the key points to be considered and which are the main challenges to overcome for achieving these results.

Section 2 reports a conceptual view of what a self-architecting autonomic microservice is, Section 3 describes the proof of concept by focusing on those elements that are relevant for this paper; Section 4 and Section 5 report discussions on some key points and challenges that can be extracted from the proof of concept; finally Section 6 contains conclusions and comments about references.

2 Self-architecting conceptual view

This section is devoted to provide a conceptual overview of what is meant here by self-architecting autonomous microservices. The discussion is kept as abstract as possible in order to illustrate only the basic concepts, focusing on the architecture of the components of the microservice application, without taking into account other details and, above all, without considering the execution context in which the microservice is deployed.

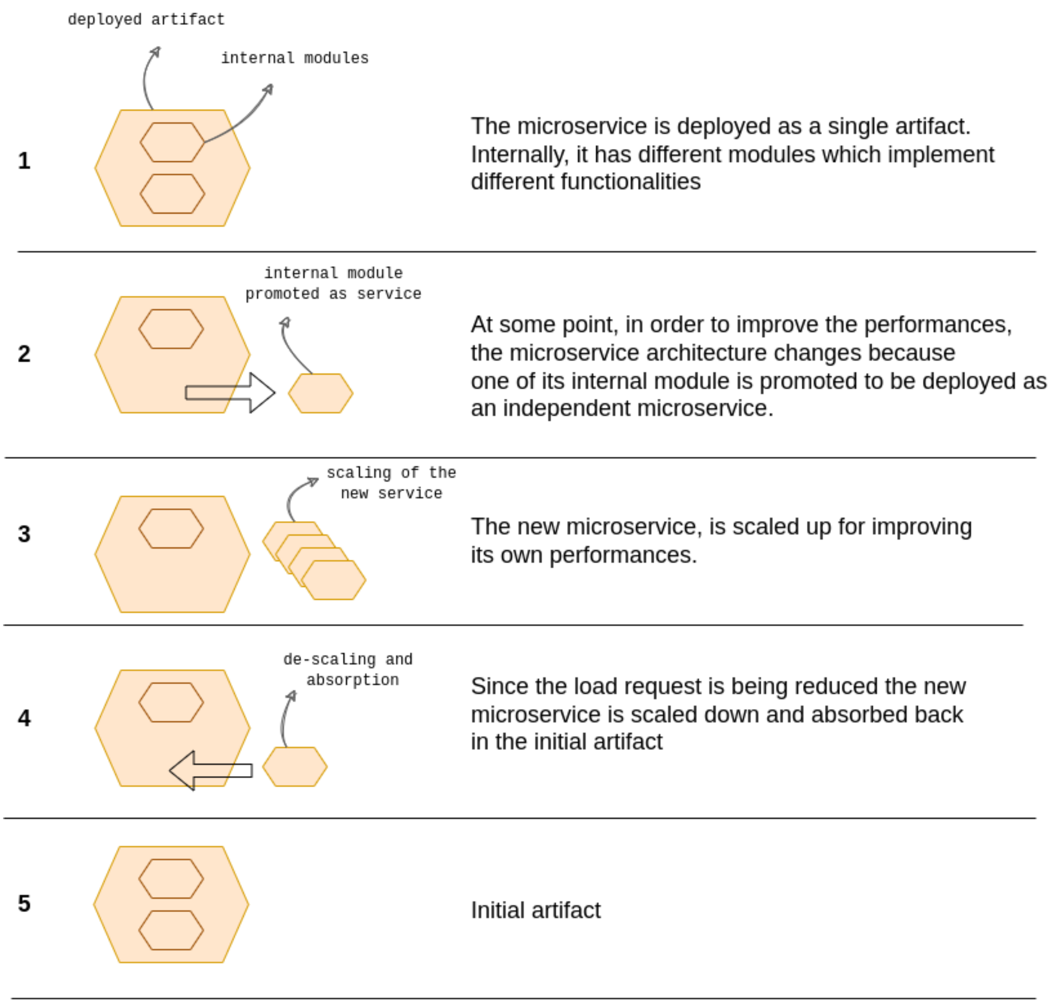


Figure 1 Expected architectural evolution of a microservice application which abstractly refers to what is going to be detailed with the proof of concept.

Such an abstraction will allow us to highlight the main contribution behind the adoption of an autonomic behaviour in modifying the architecture of a microservice application, and will make it easier to understand the description of the proof of concept that will be presented in the next section.

Figure 1 reports a conceptual architectural evolution of the microservice application targeted in the proof of concept. In Step 1, the microservice application is initially deployed: it is a single executable artifact, internally composed by different business logic modules which deal with different functionalities. In Step 2, in order to improve the performances due to an increase of load, a decision is taken: one of its internal module is promoted to become an independent microservice and it is deployed separately from the initial artifact. In Step 3, the new microservice is scaled up to specifically improve its own performances. In Step 4, since the external load is being reduced, the new microservice is scaled down and absorbed back in the initial artifact. In Step 5, the microservice is operating as it was initially.

The architectural evolution described in Figure 1 has been kept deliberately abstracted to focus on concepts about architecture modification, without specifying any actor which is responsible to perform the steps. Some questions easily emerge: in which steps there is a human intervention? In which steps does the microservice act autonomously? Moreover, which are the differences if we approach the same evolution in a conventional way, or using an autonomic approach? So far, a discussion on how such an architectural evolution could be approached and which impacts it could have on the microservice development and maintenance, has not been reported. The same evolution indeed, could be achieved following a conventional approach to the development of microservices, or using an autonomic one. A brief comparison between the conventional approach and the autonomic one, together with an analysis of the roles involved in each step, will help us to focus on better highlighting the impact of a self-architecting autonomic microservice, which is the subject of this paper. In Table 1, a comparison between the convectional approach and the autonomic one, is reported, whereas in Table 2, there is a more detailed analysis about the actors involved in each step where, for the sake of this discussion, the roles *developer* and *sysadmin* are merely indicative and they must be considered as just abstract references to two archetypal roles into an organization. A deep analysis on the impacts that an autonomic microservice could have on the different strategies adopted for managing software, is out of the scope of this paper.

As can be seen, as far as the autonomic microservice is concerned, quite all the steps are managed by the microservice itself which possesses the capability to dramatically modify its architecture, whereas in a conventional approach, all the steps involve developers, system administrators, or both. In particular, in the autonomic approach all the steps are managed by the microservice, with the exception of Step 1 that is related to the first release of the software and that is in charge to the developer in both cases. In the conventional case steps 2 and 4 are in charge to human roles, whereas Step 3, if auto-scaling feature is used, it is in charge to the execution environment. In any case, in the conventional scenario, the microservice does not take decisions or performs activities which implies a change on its own architecture, but all the actions are delegated to external actors. On the contrary, in the autonomic scenario all the actions are delegated to the microservice itself.

Table 1 Differences between a conventional approach and an autonomic one. Step 5 is reported together with Step 1 because they are equivalent.

Step	Conventional Approach	Autonomic Approach
1 (5)	(i) The developer implements the microservice in a standard way as a unique artifact, internally composing different business logic modules; (ii) The developer releases the artifact and automatically (or manually) the artifact is deployed and executed (as container or a process) into a target execution environment (e.g. a containerization layer).	(i) The developer implements the microservice as an autonomic one, by envisioning the possibility for the microservice to make some modifications about its own architecture; (ii) The developer releases the artifact and automatically (or manually) the artifact is deployed and executed (as container or a process) into a target execution environment (e.g. a containerization layer).
2	(i) The developer and the sysadmin analyze the performances of the microservice; (ii) the developer decides to divide the artifact into two by promoting one of its internal modules as a microservice. She extracts the code of the module to expunge, from the initial artifact, then she puts it into another project; (iii) the developer releases both the initial artifact, without the expunged module, and the new one; both of them are deployed replacing the former one. Optionally, the new one can be deployed together with some directives to the execution platform for auto-scaling it, if not the number of replica must be defined at deploying time.	(i-iii) The microservice auto-detects that its performance is deteriorating and it decides to promote one of its internal components as a microservice. Thus, it directly deploys the new microservice by interacting with the executing environment.
3	(i) The sysadmin analyzes the performances of the microservice; (ii) The sysadmin decides to scale up or down the new microservice, in order to tune its performances. If the auto-scaling feature has been set at the previous step, the execution platform does it automatically. Note that if it is the case of a manual intervention, such a decision should be a long-term one because it is not reasonable to manually change the number of replica day by day.	(i-ii) The microservice autonomously decides to scale up or down the new component by defining the number of current replicas by interacting with the execution environment.
4	(i) The sysadmin analyzes the performances of the microservice; (ii) The developer decides to restore the initial version of the microservice because the load is now very low and there is no need to have two microservices. Note that, in this case, usually, the developer would leave the last architecture (that of Step 3) with just one replica for the new microservice, in order to avoid the costs of a new release; (iii) The developer releases the previous artifact.	(i-iii) The microservice decides to absorb new microservice ang restoring the initial architecture.

Table 2 Detailed steps with the focus on the involved actors.

Step	Description	Conventional Approach	Autonomic Approach
1 (i)	Design and development	Developer	Developer
1 (ii)	Release and deployment	Developer	Developer
2 (i)	Analysis of the performance metrics	Developer or Sysadmin	Microservice
2 (ii)	Decision to modify the architecture, and its related implementation	Developer	Microservice
2 (iii)	Release (in case of auto-scaling, configuration of the environment)	Developer and Sysadmin	Microservice
3 (i)	Analysis of the performance metrics of the new component	Execution Environment (or Sysadmin, if auto-scaling is not set)	Microservice
3 (ii)	Decision and implementation of scaling up or down	Execution Environment (or Sysadmin, if auto-scaling is not set)	Microservice
4 (i)	Analysis of the performance metrics	Developer or Sysadmin	Microservice
4 (ii)	Decision to restore the initial version and its related implementation	Developer or Sysadmin	Microservice
4 (iii)	Deployment of the initial version	Developer	Microservice

3 Proof of concept

The main objective of the proof of concept described in this section is to show the basic mechanisms behind the implementation of an *Autonomic Microservice*, which is able to modify its own architecture depending on its own performances by negotiating it with the *Execution Environment*.

In Figure 2, a representation of the architecture developed in the proof of concept is reported. The *Autonomic Microservice* is deployed within a Docker[2] container, controlled using standard Docker API by the *Execution Environment*. Moreover, it is able to self-calculate the average response time of its own API and, depending on the results, it is able to negotiate with the *Execution Environment* a change of its architecture by scaling a specific sub-component, which takes the form of another microservice.

3.1 Architecture

Since it is a proof of concept, some assumptions have been made in order to keep it as simpler as possible:

- *Simplified model for the Autonomic Microservice*: The *Autonomic Microservice* models a microservice which implements some basic functionalities for managing a set of data, and it is assumed it implements autonomic features modelled following a MAPE-K loop[13, 20].

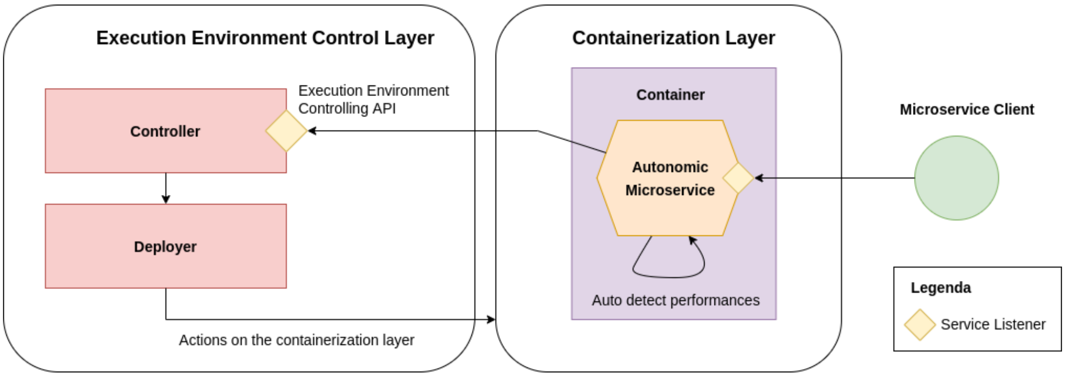


Figure 2 Logical representation of the architecture developed in the proof of concept.

In particular, all the different autonomic functionalities of the MAPE-K model have been simplified and condensed into a single internal component of the microservice. For the same reasons, all the algorithms for decision making and performance deterioration detection have been kept very basic and raw.

- *Simplified model for the Execution Environment*: the *Execution Environment* ideally models a general platform which is able to manage microservice deployment. A full representation of all of its parts is out of the scope of this paper. Here it has been modelled with a simple service that, on the one hand, it is able to interact with a containerization layer by invoking its standard API and, on the other hand, it exhibits a new set of API that are specific to be invoked by autonomic microservices in general.
- *Execution Environment agnosticism*: here we assume that the *Execution Environment* is agnostic with respect to the actual deployment an autonomic microservice may have at runtime. Apart from the first deployment, the *Execution Environment* does not own other container images nor it is aware about other components the autonomic microservice may request to have. Moreover, no specific rules for monitoring or scaling have been set in the *Execution Environment*. All the knowledge about the *Autonomic Microservice* management is in charge to the *Autonomic Microservice* itself.

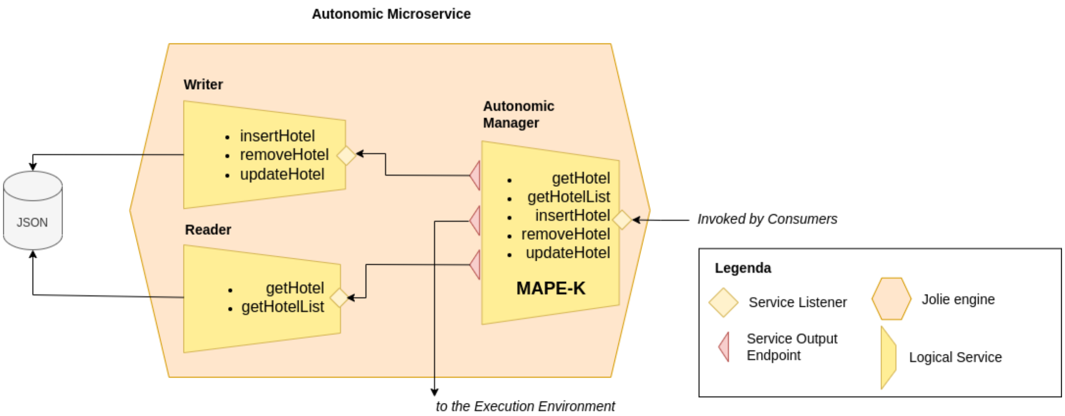


Figure 3 Autonomic Microservice inner logical architecture.

In Figure 3 the inner architecture of the *Autonomic Microservice* is reported. The microservice manages a generic set of data about a list of hotels that, for the sake of simplicity, has been modelled with a JSON[8] file[1]. Such a persistence layer is accessed by two internal services: *Writer* and *Reader*. The former is in charge to implement writing APIs (*insertHotel*, *updateHotel* and *removeHotel*), whereas the latter is in charge to implement the reading ones (*getHotel* and *getHotelList*). Neither of the two services directly exposes the APIs to the end consumer, but they are aggregated and embedded within the service *Autonomic Manager*, resulting in a single deployable artifact. A consumer can access all the APIs aggregated into the *Autonomic Manager*, by invoking its public listener where they are all available. Besides playing the role of a gateway for the reading and writing APIs listed above, the *Autonomic Manager* is also in charge to manage some autonomic features that allows for scaling the sub-service *Reader*. In particular, following a MAPE-K approach, the autonomic features of the *Autonomic Manager* can be summarized as it follows:

- *Monitor*. Since it is proxying all the requests to the inner services (*Reader* and *Writer*) it is able to capture all the metrics related to the API invocations, like invocation timestamp, reply timestamp and duration. In particular, it retrieves only those of the *Reader* because it is the component that can be scaled.
- *Analyse*. It calculates the average duration of the last ten invocation of the API of the *Reader*.
- *Plan*. It decides for a scaling or a de-scaling of the *Reader* depending on a threshold for API duration time.
- *Execute*. It interacts with the *Execution Environment* in order to ask for scaling or de-scaling the *Reader*.
- *Knowledge*. It manages the definitions of all the internal components (e.g. the *Reader*), their actual configuration (e.g. the number of active replica), and their configuration.

3.2 Runtime behaviour

In Figure 4 two scenarios, *before scaling* on the left and *after scaling* on the right, are reported. The *before scaling* represents a normal scenario where the *Autonomic Microservice* is simply deployed within a container. On the other hand, the *after scaling* represents a scenario where the *Autonomic Microservice* has been stressed with an extra load by a test consumer, and it negotiated with the *Execution Environment* for a scaling of the service *Reader*. In particular, it has been supposed that the *Autonomic Microservice* requested n instances of the service *Reader*. It is worth noting that all the instances of the service *Reader* are dynamically proxied by the *Autonomic Manager* which is in charge also to load the balance among them.

In Figure 5 the sequence chart, which describes the message exchanges between the *Autonomic Microservice* and the *Execution Environment* in case of scaling, is reported. A test client forces an extra load by continuously sending messages on the API *getHotelList* (1,2); concurrently the *Autonomic Microservice* calculate the average response time and detects a deterioration of such a metric (3). It is worth noting that, in order to trigger the scaling mechanism, the response time delay is simulated within the *Autonomic Manager* by augmenting the real measure with an extra delay. When the average time is greater than

[1] For the sake of simplicity the persistence layer has been mapped into a JSON file instead of using a structured one like a database. In the proof of concept, data consistency issues have been taken into account adding simple guards on writing and reading operations. A deep analysis about the impacts of self-architecting autonomic microservices and data consistency in a distributed scenario, is out of the scope of thsi paper.

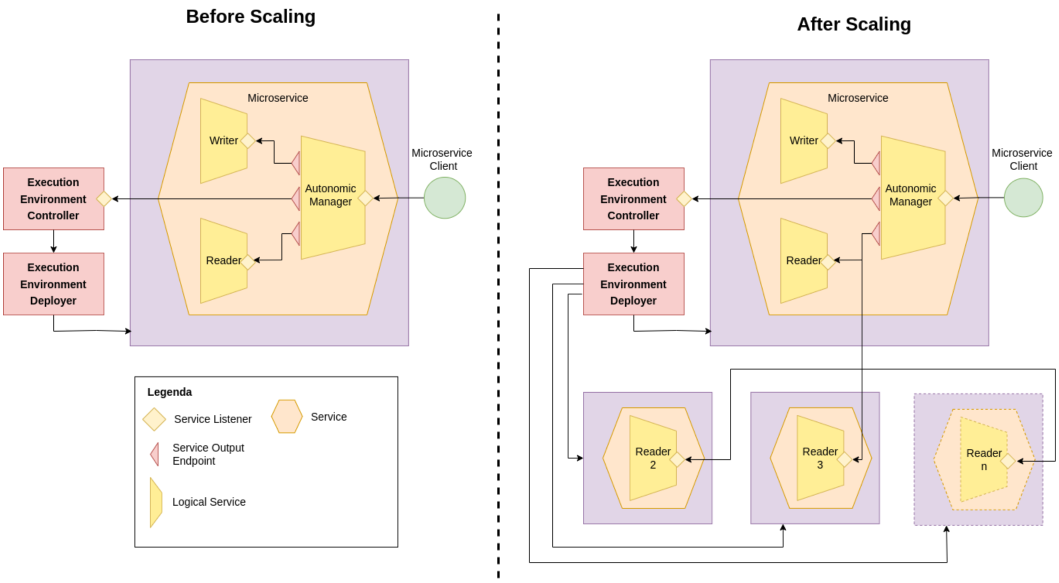


Figure 4 Microservice architecture before scaling and after scaling.

a threshold ($AVERAGE\_MAX$), the *Autonomic Manager* requests an instance of service *Reader* to the *Execution Environment* by sending also its definition (4). The *Execution Environment* creates the image for service *Reader* if it is not already stored in the internal catalogue of the containerization layer (5), and then run the container (6). Finally, it sends back the binding details of the new container to the *Autonomic Manager* (7). As a last step, the *Autonomic Manager* binds the new container into its gateway and starts to balance the load towards the new container too. Similarly, the *Autonomic Manager* can detect an improvement of the response times and request the removal of the containers that are not more necessary.

3.3 Implementation choices

The system has been realized using the service-oriented programming language Jolie[7] which has been chosen because it allows to easily implement the following aspects:

- *Embedding services*. It permits to dynamically embed a service into another. Thanks to the operator called *embedding*[19], a set of services can be executed in a distributed manner or run in the same engine. When executed within the same engine, the inner communication among the services are automatically resolved at the level of the memory without network exploitation. In the proof of concept, the *Autonomic Microservice* initially embeds both the *Writer* and the *Reader*.
- *Aggregating services*. Thanks to the operator *aggregate*[19], it permits to collect and expose APIs of different services into one single listener (in Jolie it is called *inputPort*), thus permitting to easily develop light API gateways. The aggregator plays the role of a proxy by receiving an API invocation and delivering it to the aggregated service that actually implements it. In the proof of concept, the *Autonomic Microservice* plays the role of the gateway by delivering all the incoming requests to the replica of the service *Reader*.

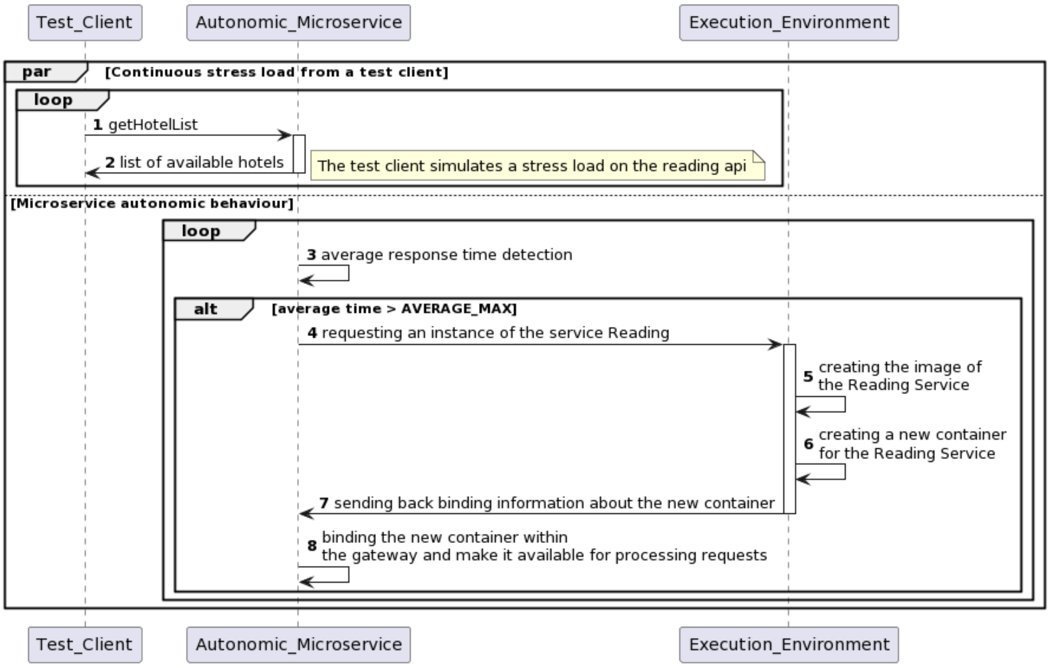


Figure 5 Sequence chart diagram for scaling.

- *Implements service functionality mobility.* It implements service functionality mobility[18] by permitting to easily send a service definition from a service to another by message. The engine which receives a service definition can dynamically embed and run it. In the proof of concept, the *Autonomic Manager* sends the definition of service *Reader* to the *Execution Environment* in order to instantiate a new replica.
- *Fast API modelling.* It permits to easily model an API using its specific syntax without following standards like openAPI[10] or gRPC[4], while preserving a comparable level of expressiveness w.r.t. them. Such a technological feature permits to reduce the technology stack burden and keep the proof of concept as simple as possible.

The code repository is public and available for inspection in [17].

4 Key points

Starting from the experience matured with the developing of the proof of concept, and following a MAPE-K loop approach, in the following some important key points that must be considered when designing and developing a self-architecting autonomic microservice, are reported:

1. **Monitor**: The *Autonomic Microservice* must collect all the required metrics for taking decisions about its own architecture that depends on the Service Level Agreements defined for that service. They can be application related, infrastructure related or both. In the former case, the microservice must be able to internally collect the metrics; in the latter case, the microservice must ask for them to the *Execution Environment*, thus implying that there are specific API for retrieving infrastructure metrics when needed. In general, the developer must be aware about the metrics to collect, and she has to know where and when they must be measured in the code.
   - In the proof of concept the *Autonomic Manager* directly retrieves the response time of each API invocation and it keeps on memory the last ten measures.

2. **Analysis**: In general, the analysis of the metrics should be condensed in few parameters calculated at runtime, thus avoiding to persist a log storage. In order to not interfere with the business logic, such a calculation should be performed concurrently. Some kind of extra volatile memory components could be introduced for persisting a limited buffer of logs. In any case, as for the monitor phase, also for the analysis, the developer must be aware of the parameters to calculate and their algorithms, and she has to design and implement a proper architecture for dealing with them.
   - In the proof of concept, after each invocation call, the *Autonomic Manager* asynchronously calculates the average duration of the last ten invocations and, depending on the result, it sends a request for a new replica of the service *Reader*.
3. **Planning**: This phase can be as complex as desired depending on the level of transformation that the service can achieve. In general, the implementation of these algorithms requires a deep knowledge about the possible evolution of the architecture. The developer must identify the degrees of freedom of the microservice and must be familiar with all the possible architectures that can be derived from it. The more degrees of freedom there are, the more the system can reach unexpected and unpredictable configurations.
   - In the proof of concept, the microservice had just one degree of freedom: it can choose to scale or remove instances of the service *Reader*. As a first glance, it looks very simple and straightforward. But it is worth noting that, in the example, there is no programmed upper limit in the algorithm. This means that the planning relies solely on the assumption that scaling the service *Reader* will eventually lead to a decrease in response times. However, if, for some reason, the variation of the response times in the real system is not strictly dependent on the number of running instances of the service *Reader*, the microservice may potentially require an infinite number of its instances, thus harming the entire system.
4. **Execution**: The *Autonomic Microservice* must implement the mechanics for transforming its own architecture. It must be able to perform the right calls to the *Execution Environment* for requesting new instances and removing the existing ones, but it also needs to provide all the components for integrating the new instances within its execution boundary.
   - In the proof of concept, the *Autonomic Manager* plays also the role of proxy and load balancer for correctly dispatching the requests to the instances of the service *Reader*. In this case, the load balancing strategy has been encoded at developing time, but it is possible to imagine making it configurable or even negotiable with the *Execution Environment*. Moreover, it is reasonable to assume that also the *Autonomic Microservice* must exhibit a set of API for being invoked by the *Execution Environment*, thus permitting a two-sided negotiation. Indeed, some architectural changes could be triggered by the environment (e.g. for optimizing the resources).
5. **Knowledge**: the *Autonomic Microservice* must manage the knowledge about its own architecture and its dynamic modification at runtime, thus it must be able to reconstruct the state of the architecture at any given time and in any condition (e.g. after a malfunctioning).
   - In the proof of concept the *Autonomic Manager* actually collects all the definitions of the components it asks to instantiate and it is able to properly configure them. But, in the current implementation, the mapping of the replicas of the service *Reader* are managed only in the volatile memory and in case of crash, the service is not able to restore such a list, thus making the existing replicas useless.

5 Main Challenges

Starting from what was outlined in the previous section, here I highlight three main challenges that need to be addressed in order to envision an engineered utilization of self-architecting autonomic microservices. In particular, these three challenges specifically focus on three different aspects: *Development activities*, *Preparation of the execution environment* and *Security management*. *Development activities* were considered because of the huge impacts autonomic computing could have on the development phase; *Adaptation of the execution environment* was considered because the adoption of an autonomic computing strategy is strongly coupled with an execution environment capable of managing it; finally *Security management* was considered because of the high level of risks that could be raised by shifting the responsibilities of many controls to the component itself.

It is important to bear in mind that the following list is not intended to cover all the critical aspects, but it is the result of a first internal evaluation about applying the approach presented in this paper, on products and applications of the author's company.

1. **Development activities:** *alleviating developer's cognitive burden.* In general, it is possible to state that the implementation of a self-architecting autonomic microservice requires an increment of the cognitive burden in charge to the developer that must be aware of all the aspects regarding the autonomic features: monitor, analysis, planning, execution and knowledge. The topic of tests deserves a special mention, because it will be necessary to test the different architectural configurations achievable by the microservice by simulating the various expected triggers and possibly mocking the execution environment, thus increasing the complexity of this task. Such a challenge could be addressed by introducing a development framework that already takes into account the various aspects necessary for the implementation of an autonomous service and partially manages them on behalf of the developer, moreover we could imagine to define the autonomic behaviour by using a specific declarative language which could help in better defining and controlling it.
2. **Adaptation of the execution environment:** *standardization of API.* In general, the *Execution Environment* should be enabled for accepting autonomic microservices, and the message exchange protocols between it and the autonomic microservice must be previously defined. Thus, the API of the *Execution Environment*, but also those that must be possibly offered by the microservice, should be standardized in order to make them equally available in any execution context where autonomic capabilities are accepted. This challenge requires a shared understanding among the developer community and, above all, among platform providers. A manifesto could be prepared and shared in order to attract valuable stakeholders for paving the way for standardization.
3. **Security management:** *security must be guaranteed by the Execution Environment.* Since an autonomic microservice is potentially able to completely change its initial architecture, thus transforming a service that it is initially safe into an harmful software artifact, the *Execution Environment* must take the responsibility to perform security checks on the autonomic microservices. In particular, the *Execution Environment* should be able to inspect the microservice and all its components before creating running instances, thus determining if the components contain malicious code. Such an aspect could be addressed by avoiding the execution of pre-compiled code, but postponing the compilation inside the *Execution Environment* and installing a microservice from sources. Constraints could be added in the allowed programming languages, thus reducing the security checks to formal ones as much as possible. As an example, languages like Ballerina[1] and Jolie[7] directly provide a linguistic tool for programming services that are then interpreted by an underlying engine that, like it happens in the proof of concept, could be directly provided by the *Execution Environment*.

6 Conclusions

The main contribution of this article is to take a step forward in the investigation of autonomous microservices capable of dynamically transforming their architecture at runtime to respond to a change in execution context. In literature there exist some general overviews about challenges and opportunity of autonomic components[15] and microservices[22] too, but a specific insight about self-architecting microservices is not reported. Other authors explored the possibility to implement autonomic microservices[25], but they focus on self-healing and versioning instead of self-architecting autonomic features.

The main benefit of the introduction of autonomic behaviours in microservices is the fully decoupling between execution environments and microservices. Such an objective is ambitious and disruptive, because it can potentially change the way microservices are developed and deployed. At the same time, however, in the long term, it could permit to reduce maintenance costs and platform's lock-in. In particular, self-architecting autonomic microservices could simplify the deployment phases because almost all the steps are delegated to the microservice. As a counterpart, issues like security, standardization and the increase of complexity on the developments side must be considered. In general new models and references are needed like in [14], where the authors propose a MAPE-K loop based reference for identifying the different responsibilities between the execution environment and the autonomic microservice.

As an evolution of this work, it could be interesting to investigate the relationship of self-architecting microservices with infrastructure as a code (IAC) approach[23] that is exploited for increasing automation in DevOps contexts. In particular, it could be interesting to apply a self-architecting behaviour over a IAC layer, thus extending the autonomic behaviour, so far restricted at the containerization level, to the infrastructure. Moreover, a non-functional decoupling between microservices and execution platforms could potentially impact internal organizational processes based on established standards, as for example ITIL[24]. Therefore, a potential area of investigation could involve analyzing how autonomic computing might influence these standards.

References

1 Ballerina. URL: `https://ballerina.io/`.
2 Docker. URL: `https://www.docker.com/`.
3 Gitlab. URL: `https://about.gitlab.com/`.
4 grpc. URL: `https://grpc.io/`.
5 Itil: Information technology infrastructure library. URL: `https://www.axelos.com/certifications/itil-service-management`.
6 Jenkins. URL: `https://www.jenkins.io/`.
7 Jolie - the service oriented programming language. URL: `https://www.jolie-lang.org`.
8 Json. URL: `https://www.json.org/`.
9 Kubernetes. URL: `https://kubernetes.io/`.
10 Openapi specification. URL: `https://swagger.io/specification/`.
11 Openshift. URL: `https://openshift.com/`.
12 Patrick debois on the state of devops. URL: `https://www.infoq.com/interviews/debois-devops/`.
13 Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-02161-9_3`.

14 Antonio Bucchiarone, Claudio Guidi, Ivan Lanese, Nelly Bencomo, and Josef Spillner. A MAPE-K approach to autonomic microservices. In *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*, pages 100–103. IEEE, 2022. doi:10.1109/ICSA-C54293.2022.00025.

15 Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009. doi:10.1007/978-3-642-02161-9_1.

16 Massimiliano Di Penta. Understanding and improving continuous integration and delivery practice using data from the wild. In *Proc. of the 13th Innovations in Software Engineering Conf. on Formerly Known as India Software Engineering Conference*, ISEC 2020, New York, NY, USA, 2020. ACM. doi:10.1145/3385032.3385059.

17 Claudio Guidi. Autonomic microservices - proof of concept code repository. URL: https://github.com/klag/autonomic-microservices.

18 Claudio Guidi and Roberto Lucchi. Formalizing mobility in service oriented computing. *J. Softw.*, 2(1):1–13, 2007. doi:10.4304/JSW.2.1.1-13.

19 Claudio Guidi and Fabrizio Montesi. Reasoning about a service-oriented programming paradigm. In Maurice H. ter Beek, editor, *Proceedings Fourth European Young Researchers Workshop on Service Oriented Computing, YR-SOC 2009, Pisa, Italy, 17-19th June 2009*, volume 2 of *EPTCS*, pages 67–81, 2009. URL: http://arxiv.org/abs/0906.3920.

20 IBM. An architectural blueprint for autonomic computing. Technical report, IBM, jun 2005.

21 Jeffrey Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, feb 2003. doi:10.1109/MC.2003.1160055.

22 Nabor Chagas Mendonça, Pooyan Jamshidi, David Garlan, and Claus Pahl. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Softw.*, 38(2):70–79, 2021. doi:10.1109/MS.2019.2955937.

23 K. Morris. *Infrastructure as Code.* O'Reilly Media, 2016.

24 Manuel Mora Tavarez, Jorge Marx Gómez, Rory V. O'Connor, Mahesh S. Raisinghani, and Ovsei Gelman. An extensive review of it service design in seven international itsm processes frameworks: Part ii. *Int. J. Inf. Technol. Syst. Approach*, 8:69–90, 2015. URL: https://api.semanticscholar.org/CorpusID:31968040, doi:10.4018/IJITSA.2015010104.

25 Yuwei Wang. Towards service discovery and autonomic version management in self-healing microservices architecture. In Laurence Duchien, Anne Koziolek, Raffaela Mirandola, Elena Maria Navarro Martínez, Clément Quinton, Riccardo Scandariato, Patrizia Scandurra, Catia Trubiani, and Danny Weyns, editors, *Proceedings of the 13th European Conference on Software Architecture, ECSA 2019, Paris, France, September 9-13, 2019, Companion Proceedings (Proceedings Volume 2),*, pages 63–66. ACM, 2019. doi:10.1145/3344948.3344952.

Correct-By-Construction Microservices with Model-Driven Engineering

The Case for Architectural Pattern Conformance Checking and Pattern-Conform Code Generation

Florian Rademacher[1]
Software Engineering, RWTH Aachen University, Germany

Abstract

Patterns are a common metaphor in software engineering that denotes reusable solutions for recurring software engineering problems. Architectural patterns focus on the interplay or organization of two or more components of a software system, and are particularly helpful in the design of complex software architectures such as those produced by Microservice Architecture (MSA).

This paper presents an approach for the language-based reification of architectural MSA patterns. To this end, we introduce a method to flexibly retrofit LEMMA (Language Ecosystem for Modeling Microservices) with support for modeling and implementing architectural MSA patterns. The method relies on the (i) specification of aspects to reify pattern applications in MSA models; (ii) validation of pattern applications; and (iii) code generation from correct pattern applications. Consequently, it contributes to correct-by-construction microservices by abstracting from the complexity of pattern implementations, yet still enabling their automated production with Model-Driven Engineering. We validate our method with the popular Domain Event and Command Query Responsibility Segregation patterns, and assess its applicability for 28 additional patterns. Our results show that LEMMA's expressivity covers the model-based expression of complex architectural MSA patterns and that its model processing facilities support pattern-specific extensions such that conformance checking and pattern-conform code generation can be modularized into reusable model processors.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Software design engineering; Software and its engineering → Source code generation; Software and its engineering → Model-driven software engineering; Software and its engineering → Software architectures; Software and its engineering → Cloud computing

Keywords and phrases Microservice Architecture, Architectural Patterns, Model-Driven Engineering, Static Model Analysis, Model Validation, Code Generation, Architectural Pattern Conformance Checking

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.8

Related Version *Previous Version*: `Microservices 2020 Extended Abstract`

Supplementary Material *Software (Language Ecosystem for Modeling Microservice Architecture (LEMMA))*: `https://github.com/SeelabFhdo/lemma`
archived at `swh:1:dir:4ac248661825a16a18a88b976734455f601e0d85`

1 Introduction

Patterns are a common metaphor in software engineering that emerged from urban design [1] and denotes reusable solutions for recurring software engineering problems [20]. Patterns are particularly useful to codify and communicate knowledge gained from software engineering experience [69, 68].

[1] This work was conducted while working at IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany.

Considering their area of impact, patterns can be divided into two major categories. *Design patterns* [20] focus on software component internals whereas *architectural patterns* [8] recognize the interplay or organization of two or more components [68]. This paper focuses on the latter category in the context of Microservice Architecture (MSA) [36]. The inception of catalogs of architectural patterns for MSA engineering [55, 70] and the study of patterns' practical adoption [35, 64] show their relevance in dealing with complexity in microservice architecture design, implementation, and operation [59]. *Language-based approaches to MSA engineering* [21] constitute a paradigm that promotes the use of MSA-oriented software languages to conceive and deploy microservice architectures, thereby coping with MSA's complexity by specialized language primitives, e.g., for microservice interfaces or containers.

This paper presents an approach that combines both means for handling complexity in MSA engineering, i.e., architectural patterns and MSA-oriented software languages. More precisely, we show how to retrofit LEMMA – an ecosystem of modeling languages and model processing facilities for MSA [49] following Model-Driven Engineering (MDE) [10] – with architectural pattern support with the goal to enable *correct-by-construction microservices* via pattern conformance checking and pattern-conform code generation. This paper revises our abstract [48] accepted at the Third International Conference on Microservices as follows:

- Introduction of a reusable method including all steps to systematically extend LEMMA with support for architectural MSA patterns.
- Consideration of code generation as a step to map LEMMA models with architectural patterns to pattern-conform, executable microservices.
- Approach validation with a case study beyond the scope of the abstract's running example.

With these contributions, we continue our line of work on the intersection of MDE and MSA by applying LEMMA's means for the model-based capturing of stakeholder-oriented concerns in MSA engineering [49], enrichment of MSA models with metadata [51], and model validation and code generation [21] to (i) integrate pattern concepts in architecture models; and (ii) ensure pattern conformance on the architecture design and implementation level. The paper specifically goes beyond previous publications [53, 54] which neither considered pattern conformance checking nor pattern-conform code generation.

The remainder of the paper is structured as follows. Section 2 provides an overview of architectural MSA patterns. Section 3 describes our method to systematically extend LEMMA with architectural pattern support. Section 4 validates our approach. Sections 5 and 6 compare our approach to related work and conclude the paper, respectively.

2 Overview of Microservice Architecture Patterns

To make the paper self-contained, we give a non-exhaustive overview of architectural MSA patterns, thereby relying on corresponding pattern catalogs [55, 70] and empirical studies [35, 64]. We structure the patterns by the categories introduced by Márquez and Astudillo [35].

Communication Patterns

Communication patterns deal with issues in microservice interaction. The API Gateway pattern [35, 55] proposes the provisioning of a component that acts as a façade to microservices' functionality, i.e., it exposes only relevant functionality to architecture-external clients. With this characteristic, the pattern gives rise to API Composition [55, 64], Gateway Routing [64], and Gateway Offloading [64]. While API Gateway requests are often synchronous, patterns like Domain Event [55] and Event Sourcing [55] concern asynchronous, event-based service interaction. The former pattern identifies domain concepts as events and the latter records event instances, e.g., for auditing purposes. Similarly, the Log Aggregator pattern [35, 55] centralizes the collection of additional log data.

Deployment Patterns

Deployment patterns focus on microservice deployment and deployment pipeline maintenance. The Sidecar pattern [55, 64] isolates and reuses functionality that crosscuts microservices, e.g., logging or configuration. Sidecar services cluster such functionality and provide business-oriented microservices with access. A more intrusive variant of this pattern is Microservice Chassis [55] which expects microservices' business logic to delegate handling of crosscutting concerns to programming frameworks. The Backend for Frontend [35, 55, 64] and Command Query Responsibility Segregation (CQRS) [55, 64] patterns provide microservice clients with interfaces for special needs. The former suggests several API Gateways, each dedicated to certain client needs or types. The CQRS pattern, on the other hand, considers a *logical microservice* to consist of (i) a physical *command-side microservice* that enables clients to change data; and (ii) one or more physical *query-side microservices* that enable clients to access data in client-specific representations. Asynchronous messaging ensures the synchronization between command-side and query-side microservices. The Database is the Service pattern [35] lets each microservice store its data in an isolated database to foster scalability.

Design Patterns

This category clusters patterns that focus on the architectural interplay of microservices, thereby regarding microservices as black boxes and allow pattern adoption at design time. We consider all *API design patterns* from the catalog of Zimmermann et al. [70] to belong to this category. The catalog organizes patterns along different dimensions in API design, e.g., Responsibility, Structure, or Quality. For example, the Processing Resource pattern (Responsibility) supports clustered exposure of application-level functionality in the form of activities or commands. The Parameter Tree pattern (Structure) concerns the tree-based design of request and response data structures that comprise containment relationships. On the Quality level, the API Key pattern [55, 70] secures API access by requiring eligible clients to provide a unique and valid *access token* for subsequent API access.

DevOps Patterns

DevOps patterns bridge between developer and operator concerns in MSA. Using the Externalized Configuration pattern [55, 64], microservices receive configuration values such as credentials or network location at runtime from specialized configuration services or stores. The Monitor patterns [35], and derivatives like Application Metrics [55], Distributed Tracing [55], Exception Tracking [55], and Health Check [35, 55], cover monitoring as a central DevOps practice [6]. With these patterns, microservices report events or status updates to a centralized server for visualization and analysis.

Migration Patterns

These patterns guide the decomposition of monoliths into microservice architectures [35]. The Strangler pattern [55, 64] suggests continuous decomposition by (i) steady migration of existing functionality to microservices; and (ii) direct realization of new functionality in microservices. The ANTI-CORRUPTION LAYER pattern [64, 55] defines façades between monolith and microservice architecture for translating between deviating data representations. Patterns like DECOMPOSE BY BUSINESS CAPABILITY (a microservice covers a business capability) and DECOMPOSE BY SUBDOMAIN (a microservice covers a part of the application domain) prescribe strategies to assign functionalities to microservices [55].

Orchestration Patterns

These patterns foster the orchestration of microservices as business process participants. The CONTAINER pattern [35, 55], which suggests to run microservices in lightweight virtualized environments [60], is a key enabler for horizontal scaling in MSA [12]. Patterns like SERVICE REGISTRY [35, 55] and SERVICE DISCOVERY [35, 55] abstract from service instances' network locations and support service interaction via logical names. It is possible to combine the SERVICE DISCOVERY pattern with the LOAD BALANCER pattern [35] to optimize request routing to services with spare processing resources. In such scenarios, the CIRCUIT BREAKER pattern [35, 55] increases reliability as it caps the number of faulty interaction attempts to prevent failure cascades. The SAGA pattern [55] increases data consistency in distributed transactions by dividing them into *local transactions* executed by dedicated microservices. If a local transaction fails, the responsible microservice invokes a *compensating transaction* for rollback and informs its predecessor in the transaction chain to also rollback.

3 A LEMMA-Based Method for Pattern-Conform Microservice Design and Implementation

This section presents our method to retrofit LEMMA with architectural MSA pattern support – including pattern conformance checking and pattern-conform code generation for correct-by-construction microservices[2]. Figure 1 shows the specification of the method.

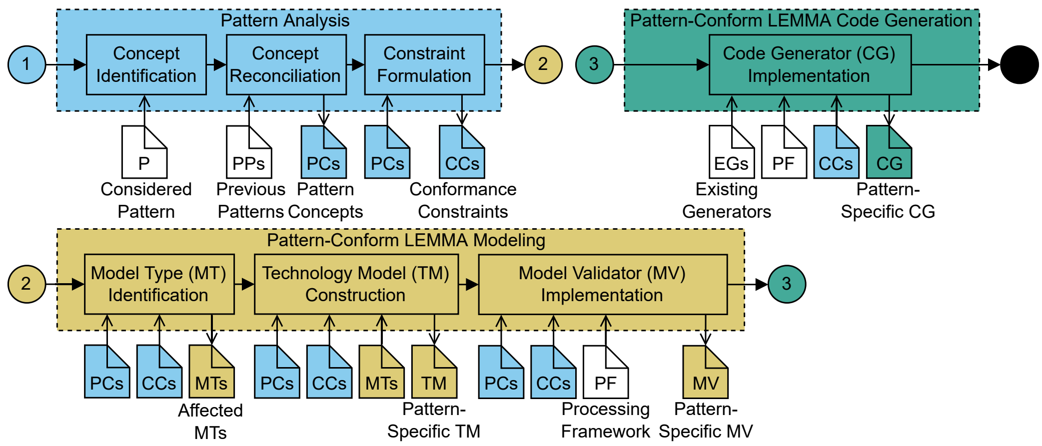


Figure 1 Overview of our method to retrofit LEMMA with architectural MSA pattern support. Method phases and activities are depicted as rectangles with dashed and solid lines, respectively. Required or produced artifacts are depicted as note sheets colored by their producing phase, if any.

[2] While we aim for the section to be understandable in a self-contained fashion, Appendices A and B yet provide additional background information on MDE and LEMMA.

Sections 3.1 to 3.3 describe the phases of the method and illustrate its instantiation on the example of the Domain Event pattern (Sect. 2).

3.1 Phase 1: Pattern Analysis

The first phase of the method consists of the three activities Concept Identification, Concept Reconciliation, and Constraint Formulation.

Concept Identification

This activity expects as input the pattern targeted by the method instance including its definition. For the Domain Event pattern, we refer to Richardson [55] who defines a domain event as a domain concept that conveys asynchronous messages between microservices.

Concept Reconciliation

The Concept Reconciliation activity expects the definitions and LEMMA-based specifications (Sect. 3.2) of all patterns covered by previous method instances as they might impact handling of the input pattern and its concepts. Suppose that the method instance for Domain Event is followed by an instance for the Event Sourcing pattern (Sect. 2). Since the latter leverages domain events, the integration of the former with LEMMA likely impacts the latter's integration. For example, it will not be necessary to treat a domain event as a genuine concept of the Event Sourcing pattern. On the other hand, the Concept Reconciliation activity helps to identify dependencies between pattern-specific LEMMA model validators (Sect. 3.2) and code generators (Sect. 3.3). For instance, the Event Sourcing pattern could require that domain events conforming to the Domain Event pattern are valid and manifested in source code so that an Event Sourcing code generator can refer to them.

The Concept Reconciliation phase yields a concept catalog for the input pattern. For Domain Event, this catalog defines the Domain Event concept as a structured domain concept representing asynchronously exchanged messages. Additionally, following the pattern's definition [55], the catalog has an entry for the Producer and Consumer concepts which denote microservices that send and receive domain events, respectively.

Constraint Formulation

The Constraint Formulation activity specifies conformance constraints for the concepts of the input pattern w.r.t. its definition. Table 1 lists constraints for the Domain Event pattern.

Table 1 Conformance constraints for the Domain Event pattern.

#	Constraint	Severity
C.1	Producer operations must be able to send asynchronous messages.	Error
C.2	Consumer operations must be able to receive asynchronous messages.	Error

Constraint C.1 prescribes that microservice operations which produce domain events must actually be able to send asynchronous messages. Constraint C.2 demands that operations of event consumers must actually be able to receive asynchronous messages. Both constraints exhibit the severity level "Error" so that their violation prevents further steps like code generation. By contrast, the severity level "Warning" would identify constraint violations that still permit subsequent processing. An example for such a violation could be a microservice operation that uses a domain event in a synchronous interaction.

3.2 Phase 2: Pattern-Conform LEMMA Modeling

This phase extends LEMMA for modeling and applying the input pattern. It involves the activities Model Type Identification, Technology Model Construction, and Model Validator Implementation.

Model Type Identification

This activity requires the pattern concepts and conformance constraints from the previous activities (Sect. 3.1) to determine the LEMMA model types (Appendix B) for pattern application. In some cases, the LEMMA modeling language for the construction of models of a certain type already comprises native constructs for pattern application. For the Domain Event pattern, LEMMA's Domain Data Modeling Language supports direct modeling of structured domain events with the `domainEvent` keyword as illustrated in Listing 1.

Listing 1 Excerpt of a LEMMA domain model illustrating domain event definition.

```
// Model name: ChargingStationManagement.data
structure ParkingAreaCreatedEvent<domainEvent> {
  immutable long commonId,
  immutable ParkingAreaInformation info
}
```

In LEMMA's Domain Data Modeling Language, a domain event is a structured domain concept that typically consists of one or more immutable fields [16]. Instances of `ParkingAreaCreatedEvent` gather a value of the primitive type `long`[3] in the field `commonId` and an instance of the structured domain concept `ParkingAreaInformation`[4] in the field `info`.

Hence, a LEMMA model type affected by extending LEMMA with Domain Event support is the domain model type because LEMMA's Domain Data Modeling Language provides native support for domain event modeling and thus the expression of the Domain Event concept from the pattern's concept catalog (Sect. 3.1). By contrast to other MDE-for-MSA approaches [2, 63, 61, 25], LEMMA does not integrate pattern-specific keywords to foster language learnability and model comprehension. As a result, LEMMA does not provide modeling constructs for Domain Event Producers and Consumers (Sect. 3.1). To retrofit these concepts, the following Technology Model Construction activity relies on the *technology aspect* mechanism [51] of LEMMA's Technology Modeling Language (Appendix B), making the technology model type also affected by LEMMA Domain Event support. Similarly, the service and mapping model types (Appendix B) are affected as both enable assignment of aspects to microservices, e.g., to mark operations as Domain Event Producers.

Technology Model Construction

LEMMA's technology aspects can be exploited as a flexible mechanism to augment model elements with metadata. We rely on such metadata to reify pattern concepts and applications in LEMMA models. Listing 2 shows LEMMA's Domain Event technology model.

---

[3] LEMMA's type system integrates all primitive types of Java [22].

[4] For the sake of brevity, we omit the specification of this domain concept and refer to the complete domain model on Software Heritage [46].

Listing 2 Domain Event technology model in LEMMA's Technology Modeling Language.

```
// Model name: DomainEvents.technology
technology DomainEvents {
  service aspects {
    aspect Producer<singleval> for operations { string handlerName<mandatory>; }
    aspect Consumer<singleval> for operations { string handlerName<mandatory>; }
  }
}
```

LEMMA supports service-related and operation-related technology aspects. The latter are applicable to operation nodes expressed in LEMMA's Operation Modeling Language (Appendix B). For the Domain Event pattern, however, we rely on service-related technology aspects which allow augmentation of LEMMA domain and service model elements. The above technology model for the Domain Event pattern specifies the `service aspects` `Producer` and `Consumer`. Both can occur at most once (`singleval`) on modeled microservice `operations` (Table 1) and require the configuration of a `handlerName`. According to the pattern definition [55], this latter property identifies the program unit, e.g., the Java class, being responsible for handling domain event production or consumption.

Listing 3 illustrates the application of the Domain Event aspects in a LEMMA service model and mapping model, respectively.

Listing 3 Excerpts of (a) a service model in LEMMA's Service Modeling Language; and (b) a mapping model in LEMMA's Service Technology Mapping Modeling Language. Both models apply the Producer concept of the Domain Event pattern based on the technology model in Listing 2.

```
1  // Model name:
2  //  ChargingStationManagementCommandSide.services
3  import datatypes from "ChargingStationManagement.data"
4    as Domain
5  import technology from "DomainEvents.technology"
6    as DomainEvents
7  @technology(DomainEvents)
8  public functional microservice
9    org.example.ChargingStationManagementCommandSide {
10   interface CommandSide {
11     @DomainEvents::_aspects.Producer
12       ("ParkingAreaProducerService")
13     sendParkingAreaCreatedEvent(
14       async out event
15         : Domain::ParkingAreaCreatedEvent
16     );
17   }
18 }
```

(a)

```
1  // Model name:
2  //  ChargingStationManagementCommandSide.mapping
3  import microservices from
4    "ChargingStationManagementCommandSide.services"
5      as CommandSideServices
6  import technology from
7    "DomainEvents.technology"
8    as DomainEvents
9  @technology(DomainEvents)
10 CommandSideServices
11   ::org.example.ChargingStationManagementCommandSide {
12   operation CommandSide.sendParkingAreaCreatedEvent {
13     aspects {
14       DomainEvents::_aspects.Producer
15         ("ParkingAreaProducerService");
16     }
17   }
18 }
```

(b)

Listing 3a shows the excerpt of a LEMMA service model that specifies a Domain Event Producer by applying the eponymous aspect from the Domain Event technology model (Listing 2). Lines 3 to 4 import the domain model that defines the `ParkingAreaCreatedEvent` domain event (Listing 1) and Lines 5 to 6 import the Domain Event technology model. Line 7 uses the built-in `@technology` annotation of LEMMA's Service Modeling Language (Appendix B) to assign the Domain Event technology model to the `ChargingStationManagementCommandSide` microservice whose definition starts in Lines 8 to 9. The service consists of the `CommandSide` interface (Line 10) that clusters the operation `sendParkingAreaCreatedEvent` (Lines 13 to 16). This operation is marked as a Domain Event Producer in Lines 11 to 12. More precisely, the assignment of the Domain Event technology model to the `ChargingStationManagementCommandSide` microservice enabled us to apply the `Producer` aspect to the operation, thereby semantically identifying it as a Domain Event Producer (Sect. 3.1).

Listing 3b also performs this semantic enrichment but on the basis of LEMMA's Service Technology Mapping Modeling Language (Appendix B). The primary difference to Listing 3a is the aspect application within the `aspects` section in Lines 13 to 16. The application of technology aspects in mapping models is an alternative to aspect application in service models.

Specifically, it externalizes technology aspect application which allows keeping service models technology-agnostic and thus reusable across different technology stacks, thereby facilitating technology migration on the model-level to deal with MSA's technology heterogeneity [36].

Model Validator Implementation

LEMMA's Technology Modeling Language already integrates certain constructs like `mandatory`, `singleval`, and the `for`-selector to constrain aspect application [51] (Listing 2). However, these constructs are not sufficient to express pattern-related constraints such as those in Table 1, e.g., because the constraints require traversal of other LEMMA models, the simultaneous application of aspects from other technology models, or certain aspect property values. We decided against the extension of the Technology Modeling Language with modeling support for such complex constraints to keep the language concise.

Instead, we employ LEMMA's Model Processing Framework (MPF; Appendix B) to accompany pattern-specific technology models with required conformance constraints. The result is a LEMMA model processor whose Model Validation phase performs constraint checking and violation reporting, and is extensible by code generation capabilities (Sect. 3.3). Listing 4 shows an excerpt of the MPF-based model validator for the Domain Event pattern written in Kotlin[5]. The complete source code can be found on Software Heritage [44].

Listing 4 Excerpt of the model validator for the Domain Event pattern in Kotlin.

```
1  $$@SourceModelValidator$$
2  internal class ServiceModelSourceValidator : AbstractXtextModelValidator() {
3    $$@Check$$
4    private fun checkProducer(operation: Operation) {
5      if (operation.hasServiceAspect("DomainEvents", "Producer") &&
6        !operation.hasResultParameters(CommunicationType.ASYNCHRONOUS))
7        error("The Producer aspect may only be applied to operations with a result " +
8          "parameter", ServicePackage.Literals.OPERATION__NAME)
9    }
10 }
```

A model validator based on the MPF must exhibit the `@SourceModelValidator` annotation [47] (Line 1) and extend the `AbstractXtextModelValidator` class [45] (Line 2). Furthermore, a validation method (Lines 4 to 9) must be preceded by the `@Check` annotation. From this annotation and the annotated method's signature, the MPF recognizes validation methods and when to invoke them. More specifically, the MPF will traverse the abstract syntax tree (AST) of a parsed LEMMA input model, identify the type of the current AST node, and, during model validation, invoke all validation methods whose signature matches this type. Consequently, it executes `checkProducer` for every modeled microservice `Operation`. In case the operation applies the `Producer` aspect from the `DomainEvents` technology model (cf. Line 5 and Listing 2) and does not have an asynchronous result parameter (Line 6), the validator yields an error using the `error` method from LEMMA's MPF (Lines 7 to 8). `checkProducer` thus implements Constraint C.1 of the Domain Event pattern (Table 1).

Model processors based on LEMMA's MPF are commandline tools that can readily be integrated into continuous integration pipelines [27]. However, the MPF also supports Live Validation for interactive model validation and error reporting. It relies on the Language Server Protocol (LSP) [11] to connect to the modeling IDE and display validation errors during model construction so that modelers need not invoke commandline validation separately from the IDE and trace errors messages to the erroneous model elements manually. Figure 2 shows the IDE manifestation of violating the check for Constraint C.1 in Listing 4.

[5] `https://www.kotlinlang.org`

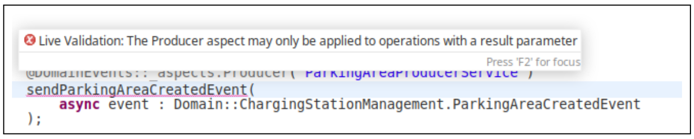


Figure 2 Model validation error resulting from an unintended application of the `Producer` aspect for the DOMAIN EVENT pattern and reported by LEMMA's MPF to the modeling IDE via the LSP.

3.3 Phase 3: Pattern-Conform LEMMA Code Generation

This phase consists solely of the Code Generator Implementation activity which has to consider possibly existing generators (Fig. 1). For example, the realization of the DOMAIN EVENT pattern requires a message broker to transmit domain events [55]. As there exist several alternative broker technologies, e.g., RabbitMQ[6] or Kafka[7], a DOMAIN EVENT code generator has to take into account the possible existence of generators for certain broker technologies. For Java-based microservices, LEMMA already bundles a Java Base Generator (JBG) that provides convenience mechanisms to map LEMMA model elements to Java code and a plugin infrastructure for MPF-based Java code generators.

In the following, we suppose that pattern-specific code generators produce Java code. Consequently, they can directly be integrated with LEMMA's JBG. Listing 5 shows an excerpt of the DOMAIN EVENT code generator plugin for the JBG. Since the JBG is based on LEMMA's MPF, this generator is part of the same model processor as the pattern's model validator (Listing 4) and its complete source code is also available on Software Heritage [43].

Listing 5 Excerpt of the DOMAIN EVENT code generator plugin for the JBG written in Kotlin.

```
1  $$@CodeGenerationHandler$$
2  internal class DataStructureHandler : GenletCodeGenerationHandlerI {
3    override fun handlesEObjectsOfInstance() = IntermediateDataStructure::class.java
4    override fun generatesNodesOfInstance() = ClassOrInterfaceDeclaration::class.java
5    override fun execute(structure: IntermediateDataStructure,
6      clazz: ClassOrInterfaceDeclaration) {
7      val eventGroup = clazz.getAspectProperty("EventGroup", "name") ?: return null
8      val groupInterface = EventGroups.addOrGetGroupInterface(eventGroup)
9      node.addImplementedType(groupInterface.nameAsString)
10     return node
11   }
12 }
```

The JBG structures the MPF's Code Generation phase (Appendix B) into code generation handlers each of which maps a particular LEMMA model element type to a Java AST node type [22]. A code generation handler is a class augmented with the `@CodeGenerationHandler` annotation (Line 1) and implementing the `GenletCodeGenerationHandlerI` interface (Line 2) such that the handler must override the methods `handlesEObjectsOfInstance` and `generatesNodesOfInstance` (Lines 3 and 4). The methods respectively inform the JBG which LEMMA model element type the handler can process and to which Java AST node type the element type corresponds. The handler in Listing 5 maps data structures in LEMMA's Domain Data Modeling Language (Appendix B) to Java class declarations.

The `execute` method (Lines 5 to 11) clusters the handler's logic. It concerns domain event grouping, and, following the pattern's definition [55], produces a Java interface for each event group which is then assigned as an implemented type to the domain event Java class.

[6] `https://www.rabbitmq.com`
[7] `https://kafka.apache.org`

4 Validation

We validate our method (Sect. 3) with CQRS, i.e., one of the most complex patterns in Sect. 2 that (i) spans several LEMMA viewpoints (Appendix B); (ii) requires more than one model for the same viewpoint and thus ad hoc model parsing; (iii) combines technology-agnostic and technology-specific validation; and (iv) impacts service configuration and source code.

We present the research questions (RQs; Sect. 4.1), case study (Sect. 4.2), and results (Sect. 4.3) of our method's validation, and discuss the latter (Sect. 4.4).

4.1 Research Questions

Our validation focuses on the following RQs:

RQ 1 To what extent do LEMMA and our method for its extension with pattern support cover patterns whose complexity exceeds that of DOMAIN EVENT significantly?

RQ 2 How much effort is required to provide such complex extensions?

4.2 Case Study

We validate our method with a case study microservice architecture called Park and Charge Platform (PACP). The PACP originates from a research project that aims to enable the offering of private charging stations for electric vehicles for use by other owners of electric vehicles. We described the PACP's architecture in more detail in a previous publication [53]. Almost all PACP microservices apply CQRS. Figure 3 provides an overview of the pattern.

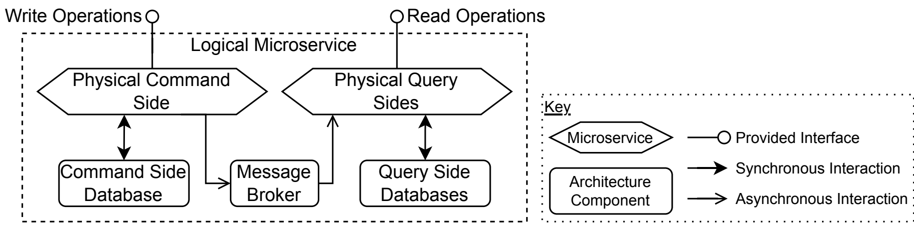


Figure 3 Overview of the CQRS pattern.

As mentioned in Sect. 2, a CQRS application consists of a logical microservice that provides clients with write and read operations. Depending on an operation's kind, its actual implementation resides in a physical command side microservice (write) or one of potentially several query side microservices (read). Command side microservices allow the alteration of data in command side databases [55] and asynchronously inform query side microservices about alterations via message brokers. Query sides then update their databases accordingly.

The benefits of CQRS for the PACP are twofold. First, we can scale read operations independently of write operations and for the PACP the former are much more frequent. Second, the pattern allows storage optimization for read operations so that we can preprocess sensor data from charging stations for time series processing as well as for relational queries.

In the following, we focus on the PACP's Charging Station Management Microservice. It is responsible for managing charging station information like location, charging and plug type, and also receives data from charging stations. While we consider only a single PACP microservice, our results are directly transferable to the CQRS-conform modeling and code generation of all other PACP microservices.

4.3 Results

We structure the presentation of the validation results by the phases of our method (Sect. 3).

Phase 1: Pattern Analysis

Figure 3 is a direct outcome of the Concept Identification activity. Structured by LEMMA's viewpoints (Appendix B), a CQRS application consists of the following concepts:

- **Domain Viewpoint:** Domain concepts to be stored in databases and used to convey asynchronous messages between physical microservices.
- **Service Viewpoint:** Logical CQRS microservice consisting of exactly one command side and at least one query side microservice. Physical microservices provide synchronous operations to consumers and interact with each other asynchronously.
- **Operation Viewpoint:** Infrastructure nodes like databases and a message broker.

In the following, we focus on the Domain and Service Viewpoint because they have the most impact on the upcoming activities.

During the Concept Reconciliation activity (Fig. 1), we not only discovered the above concepts but also the close relationship between the DOMAIN EVENT and CQRS patterns. Consequently, we decided to rely on domain events to define the structures for the asynchronous messages sent from command side to query side microservices.

Table 2 lists the constraints resulting from the Constraint Formulation activity.

Table 2 Conformance constraints for the CQRS pattern.

#	Constraint	Severity
C.3	Logical microservice consists of command and at least one query side microservice.	Error
C.4	Command side microservice should be able to send domain events.	Warning
C.5	Query side microservice should be able to receive domain events.	Warning
C.6	Incoming domain event parameters of query side operations should be type-compatible with outgoing domain event parameters of command side operations.	Warning

Violations of Constraints C.4 to C.6 yield warnings to permit iterative CQRS modeling.

Phase 2: Pattern-Conform LEMMA Modeling

From the concerned LEMMA viewpoints, the Model Type Identification activity identified the domain, service, and operation model types (Appendix B) to be affected by CQRS.

Technology Model Construction resulted in the LEMMA technology model in Listing 6.

Listing 6 CQRS technology model in LEMMA's Technology Modeling Language.

```
// Model name: Cqrs.technology
technology CQRS {
  service aspects {
    aspect CommandSide for microservices { string logicalService; }
    aspect QuerySide for microservices { string logicalService; }
  }
}
```

The `CommandSide` and `QuerySide` aspects allow determination of command side and query side microservices. Two microservices that apply these aspects belong to the same logical CQRS microservice when the values for the `logicalService` property are equal.

In the Model Validator Implementation activity, we developed a LEMMA model validator for the above technology model. Its Kotlin-based implementation consists of 180 lines of code (LOC), excluding empty lines, and is available on Software Heritage [42]. The validator implements Constraints C.4 and C.5 (Table 2).

To realize Constraint C.3, we decided to rely on a built-in construct of LEMMA's Service Modeling Language (Appendix B), i.e., the `required microservices` directive. It allows modeling of relationships between microservices in the same or distinct service models – in the latter case based on LEMMA's import mechanism and inter-model references. Hence, our CQRS validator checks the constraints in Table 2 only when a query side requires a command side microservice of the same logical microservice as identified by the `logicalService` property of the corresponding aspect applications (Listing 6). The specification of the dependency of a query side on a command side microservice is in line with CQRS because the former relies on events received by the latter to update its database accordingly (Fig. 3).

Concerning Constraint C.6, we decided against its inclusion in the CQRS validator. Instead, we extended an implementation for the type-checking of sending and receiving microservice operations in an existing JBG plugin (Sect. 3.3) for the Kafka message broker.

Phase 3: Pattern-Conform LEMMA Code Generation

LEMMA bundles a set of JBG plugins for popular MSA technologies like Kafka and Spring[8]. Among others, the Kafka plugin [40] supports the extension of Java codebases produced from LEMMA models by the JBG, e.g., with configuration code for the connection to a Kafka broker and with methods for sending events via this broker. For the extension of LEMMA with CQRS, we were able to reuse the plugin for the most part. As described above, we only had to extend its validator with a check for Constraint C.6 (Table 2). This extension concerned 141 LOC between Lines 185 and 343 of the validator's Kotlin implementation [41].

4.4 Discussion

We discuss the validation of our method w.r.t. the framed RQs (Sect. 4.1).

RQ 1 Method Applicability for Complex Architectural Patterns

Based on the successful application of the method on the CQRS pattern (Sect. 4.3) we conclude that it is applicable not only to comparatively simple patterns like Domain Event (Sect. 3) but also to patterns that involve more concepts, validations, and code to be generated. However, an in-depth evaluation of the method's applicability would require its usage on ideally all patterns from Sect. 2. In order to anticipate the structure of such an evaluation, we assessed the complexity of extending LEMMA with pattern support using our method for all of these patterns, except for Domain Event and CQRS. Table 3 shows an excerpt of this assessment for one pattern from each category in Sect. 2, besides Migration. That is, because Migration is mostly out of LEMMA's scope. We refer to Appendix C for the complete list of assessed pattern extension complexity.

API Gateway is the example of a pattern that requires comparatively low effort to be integrated with LEMMA. That is because it concerns only the Operation viewpoint (Appendix B) and requires the modeling of one infrastructure node that needs to be augmented with the technology for an API gateway. The amount of generated code is probably also

[8] `https://www.spring.io`

Table 3 Assessed complexity of extending LEMMA with support for selected architectural MSA patterns (see Appendix C for the complete list).

Pattern	LEMMA Viewpoints			Quantities a)			Extension Complexity b)
	Dom.	Serv.	Op.	Concepts	Constraints	LOC c)	
API Gateway			✓	○	○	○	○
Backend for Frontend	✓	✓	✓	○	○	●	◐
API Key	✓	✓	✓	○	◐	●	●
Distributed Tracing		✓	✓	●	●	●	●
Container			✓	○	○	●	N/A

Symbol key: ○ = Low; ◐ = Middle; ● = High.
a) Assessed relative across all patterns.
b) Assessed based on quantities, and experiences with Domain Event (Sect. 3) and CQRS (Sect. 4).
c) Assessed quantity of generated code.

low as there exist readily usable implementations of the pattern, e.g., Zuul[9]. Similarly, the assumed integration effort for the Backed for Frontend pattern is rather low. In fact, LEMMA already provides all required concepts for modeling this pattern and, as for CQRS, only a technology model to link a backend with a frontend service will be needed. Nonetheless, the amount of generated code will likely be high since backend services are full-fledged microservices. The API Key pattern, on the other hand, calls for additional constraints that allow checking whether the additional security infrastructure was configured correctly. Moreover, we expect the necessity to generate this infrastructure to further increase the effort for API Key integration. The integration of the Distributed Tracing pattern will be nearly infeasible as it requires concepts and corresponding constraints for service interactions. However, LEMMA does currently not exhibit such concepts and the technology aspect mechanism is likely not sophisticated enough to fully retrofit them. Concerning the Container pattern, no LEMMA integration is needed as it is already fully supported by LEMMA's Operation Modeling Language (Appendix B).

RQ 2 Method Effort for Complex Architectural Patterns

Given our experience with the conceptual and implementation subtleties of LEMMA and the CQRS pattern, the execution of the method was straightforward. At least for the Pattern Analysis phase (Sect. 3), we expect the method to not impose significant overhead, even for stakeholders that are not familiar with LEMMA. That is because the phase does not require LEMMA knowledge and a basic knowledge of the targeted pattern can be assumed as otherwise the objective decision to extend LEMMA with support for it would not have been possible at all. For the first two activities of the Pattern-Conform LEMMA Modeling phase, basic LEMMA knowledge is necessary to identify the model types affected by pattern extension and construct the corresponding technology model. However, the amount of LEMMA knowledge increases significantly for subsequent model validator and code generator

[9] `https://github.com/Netflix/zuul`

development. Moreover, both these activities require additional implementation effort. LEMMA's extension with the CQRS pattern comprises 327 LOC including the technology model, validator, and extension of the Kafka JBG plugin (Sect. 4.2). For complex patterns like Gateway Offloading or Microservice Chassis (Appendix C) the LOC count is likely to increase, especially when no JBG plugin or model processor for extension exists. We consider the objective quantification of the effort required by our method of high interest for future research works.

5 Related Work

We present work related to MDE-for-MSA and architectural pattern conformance assurance.

MDE-for-MSA

There exists a plethora of MDE approaches with support for the MSA viewpoints Domain [13, 63, 29], Service [23, 2, 61, 29, 25], and Operation [18, 2, 61, 25] (Appendix B). By contrast to LEMMA and the presented method (Sect. 3), the majority of these approaches does not allow pattern integration on the language-level. Only DCSL [13] can be considered to have basic support for language-level extensibility leveraging meta-attributes. However, it only covers domain modeling and only a subset of architectural MSA patterns concerns the Domain viewpoint (Appendix C). Instead, most of the patterns are rooted in the Service and Operation viewpoints. As opposed to LEMMA, MDE-for-MSA approaches for these viewpoints integrate pattern-specific concepts like `Circuit Breaker` [2], `APIGatewayService` [61], or `serviceDiscoveryType` [25] into provided modeling languages. Hence, these approaches aggravate learnability by extensive language syntaxes and prevent retrofitting, thus requiring new releases each time the modeling of a new pattern shall be supported.

Architectural Pattern Conformance Assurance

Conformance checking between the intended design of a software and its actual implementation is an important activity in software architecting [5] that may also reveal deviations from pattern specifications. Kim and Shen [30] leverage a divide-and-conquer strategy to assess the syntactic conformance of UML class diagrams [39] with design patterns that are specified via a UML extension for role-based metamodeling. Roles in pattern models prescribe contributions to pattern solutions and can be played by more than one element in assessed class diagrams. Consequently, the approach is able to also capture pattern variations. Chihada et al. [9] present an approach to pattern conformance checking based on Support Vector Machines. To this end, classifiers for design patterns are trained based on the peculiarities of object-oriented metrics in pattern specifications. In the next step, the classifiers are applied to source code that has been partitioned into smaller chunks of possible design pattern manifestations. From the confidence values returned by classifiers, Chihada et al. assess the likeliness for pattern occurrence. This approach is also able to capture pattern variations. However, other than Kim and Shen, Chihada et al. operate on source code. Díaz-Pace et al. [14] suggest a heuristic approach to detect source code deviations from behavioral architecture scenarios expressed as use-case maps (UCMs) which are graph-based descriptions of expected system executions. Deviations between UCM specifications and evolved source code are then detected by exercising the architecture implementation against test cases derived from UCMs. Similar to Kim and Shen, Díaz-Pace et al. identify conformance by means of an abstract view on component responsibilities. By contrast to the aforementioned works, our method

(Sect. 3) aims to ensure pattern conformance by model validation and code generation. While this approach allows pattern expression in the same constructive source model and on a level of abstraction that facilitates pattern recognition, it also expects the production of code that is actually pattern-conform, thereby bundling pattern conformance checking and pattern-conform code generation in model processors. Furthermore, we focus on architectural patterns instead of design patterns and do not explicitly consider pattern variations. They may however be codified as aspect properties in pattern technology models so that code generators produce varied pattern instances w.r.t. property values. Similar to Chihada et al., we also consider pattern analysis a crucial step prior to pattern handling.

There also exist works concerning the assessment and restoration of architectural pattern conformance in MSA engineering [38, 37]. These works are of particular interest to us because they define metrics and refactorings on the model-level to determine and resolve pattern deviations. We regard the integration of these approaches with our method as a sensible future work as it would permit (i) testing the actual conformance of generated pattern code, thereby enabling model processor developers to evaluate processor correctness; and (ii) resolution of pattern deviations from reconstructed architecture models [52].

Similar to us, Falkenthal et al. [17] focus on linking pattern specifications with solution implementations which may be source code that reifies a pattern realization. Next to pattern variation, the work by Falkenthal et al. raises an important concern regarding the modularization and composition of pattern solutions. Our method recognizes this concern in Phase 3 (Fig. 1), where existing code generators are examined prior to the implementation of a novel pattern-specific code generator in order to identify generator dependencies and reusability. In its current form, this examination has to be conducted manually, and thus requires deep knowledge about existing generators and their implementations. It would therefore be beneficial to reason about means to describe code generator composability on a more abstract level, e.g., by extending LEMMA's Technology Modeling Language with capabilities to express composition relationships between pattern-specific technology aspects.

6 Conclusion and Future Work

In this paper, we presented an approach that combines two orthogonal means for complexity reduction in Microservice Architecture (MSA) engineering, i.e., architectural MSA patterns (Sect. 2) and language-based approaches to MSA. More precisely, we introduced a method (Sect. 3) for the systematic retrofitting of LEMMA (Language Ecosystem for Modeling Microservices) with support for modeling and implementing architectural MSA patterns. The presented method relies on the (i) specification of aspects to reify pattern applications in MSA models; (ii) validation of pattern applications for correctness; and (iii) code generation from MSA models with correct pattern applications. We validated the feasibility of our method for two popular architectural MSA patterns, i.e., Domain Event and Command Query Responsibility Segregation (CQRS), and provide a complexity assessment for the integration of 28 other patterns with LEMMA (Sect. 4). Our approach aims at enabling correct-by-construction microservices by abstracting from the complexity of pattern implementations, yet still enabling their automated production with techniques from Model-Driven Engineering (MDE). In particular, our results show that LEMMA's expressivity is versatile enough to even allow the model-based expression of comparatively complex architectural MSA patterns such as CQRS and that the provided model processing facilities support pattern-specific LEMMA extensions such that pattern conformance checking and pattern-conform code generation can be modularized into reusable model processors.

While we are confident that from the 28 remaining architectural MSA patterns, 19 exhibit a low to middle complexity concerning their integration with LEMMA, an objective quantification is still necessary. We therefore plan to extend LEMMA with support for all of these patterns and identify even further applicable patterns, e.g., from Cloud Computing [62]. We are also interested in applying our approach with other works targeting the model-based assessment and restoration of architectural pattern conformance in microservice implementations. As a result, it would become possible to detect pattern applications and deviations from source code, and exploit the presented approach for the correct resolution of such deviations. We are also interested in extending LEMMA with means to anticipate the composability of pattern-specific code generators, e.g., by expressing composition relationships between pattern-specific technology aspects on the model-level.

References

1 Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

2 Nuha Alshuqayran, Nour Ali, and Roger Evans. Towards micro service architecture recovery: An empirical study. In *2018 Int. Conf. on Softw. Architecture (ICSA)*, pages 47–56. IEEE, 2018. `doi:10.1109/ICSA.2018.00014`.

3 David Ameller, Xavier Burgués, Oriol Collell, Dolors Costal, Xavier Franch, and Mike P. Papazoglou. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, 62:42–66, 2015. Elsevier. `doi:10.1016/J.INFSOF.2015.02.006`.

4 Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a large industrial context—Motorola case study. In *Model Driven Engineering Languages and Systems*, pages 476–491, Berlin, Heidelberg, 2005. Springer. `doi:10.1007/11557432_36`.

5 Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, third edition, 2013.

6 Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Pearson, 2015. URL: `http://bookshop.pearson.de/devops.html?productid=208463`.

7 Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 Int. Conf. on Softw. Architecture Companion (ICSA-C)*, pages 187–195. IEEE, 2019. `doi:10.1109/ICSA-C.2019.00041`.

8 Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-Oriented Software Architecture*, volume 5. Wiley, 2007. URL: `https://www.worldcat.org/oclc/314792015`.

9 Abdullah Chihada, Saeed Jalili, Seyed Mohammad Hossein Hasheminejad, and Mohammad Hossein Zangooei. Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing*, 26:357–367, 2015. `doi:10.1016/J.ASOC.2014.10.027`.

10 Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC, 2017.

11 Microsoft Corporation. Language Server Protocol specification - 3.17, 2022. URL: `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification`.

12 Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How to make your application scale. In *Perspectives of System Informatics*, pages 95–104. Springer, 2018. `doi:10.1007/978-3-319-74313-4_8`.

13 Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. Domain-driven design using meta-attributes: A DSL-based approach. In *2016 Eighth International Conference on Knowledge and Systems Engineering (KSE)*, pages 67–72. IEEE, 2016. `doi:10.1109/KSE.2016.7758031`.

14 J.A. Díaz-Pace, Álvaro Soria, Guillermo Rodríguez, and Marcelo R. Campo. Assisting conformance checks between architectural scenarios and implementation. *Information and Software Technology*, 54(5):448–466, 2012. doi:10.1016/J.INFSOF.2011.12.005.

15 Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

16 Eric Evans. *Domain-Driven Design Reference*. Dog Ear Publishing, 2015.

17 Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, and Frank Leymann. From pattern languages to solution implementations. In *Proceedings of the 6th International Conference on Pervasive Patterns and Applications (PATTERNS 2014)*, pages 12–21. Xpert Publishing Services (XPS), 2014.

18 Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894. IEEE, 2013. doi:10.1109/CLOUD.2013.133.

19 Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering (FOSE)*, pages 37–54. IEEE, 2007. doi:10.1109/FOSE.2007.14.

20 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

21 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, and Florian Rademacher. Model-driven generation of microservice interfaces: From LEMMA domain models to Jolie APIs. In *Coordination Models and Languages*, pages 223–240. Springer, 2022. doi:10.1007/978-3-031-08143-9_13.

22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java language specification: Java SE 17 edition. JSR-392, Oracle Inc., 2021.

23 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 Int. Conf. on Softw. Arch. Workshops (ICSAW)*, pages 46–53. IEEE, 2017. doi:10.1109/ICSAW.2017.48.

24 ISO/IEC/IEEE. Systems and software engineering — Architecture description, 2011.

25 JHipster. Jhipster domain language (jdl), 2022-14-02. URL: https://www.jhipster.tech/jdl.

26 Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. SIGS Publications.

27 Robbert Jongeling, Jan Carlson, and Antonio Cicchetti. Impediments to introducing continuous integration for model-based development in industry. In *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 434–441. IEEE, 2019. doi:10.1109/SEAA.2019.00071.

28 Stefan Kapferer and Olaf Zimmermann. Domain-driven service design. In *Service-Oriented Computing*, pages 189–208. Springer, 2020. doi:10.1007/978-3-030-64846-6_11.

29 Stefan Kapferer and Olaf Zimmermann. Domain-specific language and tools for strategic Domain-driven Design, context mapping and bounded context modeling. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, pages 299–306. INSTICC, SciTePress, 2020. doi:10.5220/0008910502990306.

30 Dae-Kyoo Kim and Wuwei Shen. Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 16(3):329–359, sep 2008. doi:10.1007/S11219-008-9048-5.

31 Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Montithings: Model-driven development and deployment of reliable iot applications. *Journal of Systems and Software*, 183:111087, 2022. doi:10.1016/J.JSS.2021.111087.

32 Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014. IEEE. doi:10.1109/TSE.2014.2354344.

33 Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - A review of experiences from applying MDE in industry. In *Model Driven Architecture – Foundations and Applications*, pages 432–443. Springer, 2008. doi:10.1007/978-3-540-69100-6_31.

34 Mustafa Abshir Mohamed, Moharram Challenger, and Geylani Kardas. Applications of model-driven engineering in cyber-physical systems: A systematic mapping study. *Journal of Computer Languages*, 59:1–54, 2020. doi:10.1016/J.COLA.2020.100972.

35 Gastón Márquez and Hernán Astudillo. Actual use of architectural patterns in microservices-based open source projects. In *25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. IEEE, 2018. doi:10.1109/APSEC.2018.00017.

36 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015. URL: https://www.worldcat.org/oclc/904463848.

37 Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, and Sebastian Geiger. Evaluating and improving microservice architecture conformance to architectural design decisions. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing*, pages 188–203, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-91431-8_12.

38 Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari, editors, *Service-Oriented Computing*, pages 580–596, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-65310-1_42.

39 OMG. OMG Unified Modeling Language (OMG UML) version 2.5.1. Standard formal/17-12-05, Object Management Group, 2017.

40 Florian Rademacher. Kafka plugin for LEMMA's Java Base Generator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:dir:98fdbe5eb42ef33cf9cc7895ef47f6ca8f3791b0;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%252520generators/de.fhdo.lemma.model_processing.code_generation.springcloud.kafka/.

41 Florian Rademacher. Service model validator in the kafka plugin for LEMMA's Java Base Generator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:3b1e8bb4a5da879b05b06812c855e50572bbdc96;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.kafka/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/kafka/validators/ServiceModelSourceValidator.kt.

42 Florian Rademacher. CQRS model validator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:9558df1409cbac539118ff3ed8eab013e8d84a44;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.cqrs/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/cqrs/validators/ServiceModelSourceValidator.kt.

43 Florian Rademacher. Domain Event code generation handler on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:dab6b250ef32a4da914ffdcae0b8f7f5c79e09a1;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.domain_events/src/

main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/domain_events/handlers/DataStructureHandler.kt.

44 Florian Rademacher. Domain Event model validator on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:46968c76c89ffe9cb23d480403eafcc837a4cd23;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/code%20generators/de.fhdo.lemma.model_processing.code_generation.springcloud.domain_events/src/main/kotlin/de/fhdo/lemma/model_processing/code_generation/springcloud/domain_events/validators/ServiceModelSourceValidator.kt.

45 Florian Rademacher. AbstractXtextModelValidator class of lemma's model processing framework on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:cbc0b6283dc88cee0c747f0824cf697d05db8506;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/de.fhdo.lemma.model_processing/src/main/kotlin/de/fhdo/lemma/model_processing/phases/validation/AbstractXtextModelValidator.kt.

46 Florian Rademacher. ChargingStationManagement.data lemma domain model on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:1578bbef5fb1b6463db33582c06d930236ed8653;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/examples/charging-station-management/models/domain/ChargingStationManagement.data.

47 Florian Rademacher. SourceModelValidator annotation of lemma's model processing framework on Software Heritage. URL: https://archive.softwareheritage.org/swh:1:cnt:2207b7f3f5ddbee2bda1ec9ae1968b9fed03947f;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:a668854675d50d3f3cad56eb5e3961b683d0f70a;anchor=swh:1:rev:2e9ccc882352116b253a7700b5ecf2c9316a5829;path=/de.fhdo.lemma.model_processing/src/main/kotlin/de/fhdo/lemma/model_processing/annotations/SourceModelValidator.kt.

48 Florian Rademacher. A non-intrusive approach to extend microservice modeling languages with architecture pattern support. In *Third Int. Conf. on Microservices (Microservices 2020)*, 2020. URL: https://www.conf-micro.services/2020/papers/paper_3.pdf.

49 Florian Rademacher. *A Language Ecosystem for Modeling Microservice Architecture*. PhD thesis, University of Kassel, 2022. URL: https://kobra.uni-kassel.de/handle/123456789/14176.

50 Florian Rademacher, Martin Peters, and Sabine Sachweh. Design of a domain-specific language based on a technology-independent web service framework. In *Software Architecture*, pages 357–371. Springer, 2015. doi:10.1007/978-3-319-23727-5_29.

51 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Aspect-oriented modeling of technology heterogeneity in Microservice Architecture. In *2019 Int. Conf. on Softw. Architecture (ICSA)*, pages 21–30. IEEE, 2019. doi:10.1109/ICSA.2019.00011.

52 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020. doi:10.1007/978-3-030-49418-6_21.

53 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards holistic modeling of microservice architectures using LEMMA. In *Companion Proc. of the 15th European Conf. on Software Architecture 2021*, pages 1–10. CEUR-WS, 2021. URL: http://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf.

54 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards an extensible approach for generative microservice development and deployment using lemma. In

Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, *Software Architecture*, pages 257–280. Springer, 2022. doi:10.1007/978-3-031-15116-3_12.

55 Chris Richardson. *Microservices Patterns.* Manning Publications, 2019.

56 Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation ADLs through MDE techniques. In *32nd Int. Conf. on Software Engineering (ICSE)*, volume 1, pages 85–94. IEEE, 2010. doi:10.1145/1806799.1806816.

57 Gerald Schermann, Jürgen Cito, and Philipp Leitner. All the services large and micro: Revisiting industrial practice in services computing. In *Service-Oriented Computing – ICSOC 2015 Workshops*, pages 36–47. Springer, 2016. doi:10.1007/978-3-662-50539-7_4.

58 Stefan Sobernig and Uwe Zdun. Inversion-of-Control layer. In *Proc. of the 15th European Conf. on Pattern Languages of Programs*, pages 1–22. ACM, 2010. doi:10.1145/2328909.2328935.

59 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*, 146:215–232, 2018. Elsevier. doi:10.1016/J.JSS.2018.09.082.

60 Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. of the 2nd Europ. Conf. on Computer Systems 2007*, pages 275–287. ACM, 2007. doi:10.1145/1272996.1273025.

61 Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. AjiL: Enabling model-driven microservice development. In *Proc. of the 12th Europ. Conf. on Softw. Architecture: Companion Proceedings*, pages 1:1–1:4. ACM, 2018. doi:10.1145/3241403.3241406.

62 Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 48(6):2128–2140, 2022. doi:10.1109/TSE.2021.3052177.

63 Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. *Enterprise Information Systems*, 12:1034–1057, 2018. doi:10.1080/17517575.2018.1460766.

64 Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *19th Int. Conf. on Softw. Architecture (ICSA)*, pages 69–79, 2022. doi:10.1109/ICSA53651.2022.00015.

65 Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (bcool). In *18th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195, 2015. doi:10.1109/MODELS.2015.7338249.

66 Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014. IEEE. doi:10.1109/MS.2013.65.

67 Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling languages in industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, jan 2020. doi:10.1007/S10270-019-00757-6.

68 Cheng Zhang and David Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012. doi:10.1109/TSE.2011.79.

69 Cheng Zhang and David Budgen. A survey of experienced user perceptions about software design patterns. *Information and Software Technology*, 55(5):822–835, 2013. doi:10.1016/J.INFSOF.2012.11.003.

70 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2022.

Appendix

A Model-Driven Engineering

MDE is a paradigm that promotes the use of *models* in the software engineering process [10]. In the sense of MDE, a model is a software artifact that describes selected aspects of a software system in an abstracted fashion and can replace more concrete artifacts for certain purposes. For instance, a model may be a partial representative for source code focusing on capturing component interfaces [50] or coordination [65]. A major goal of MDE is to increase models' usage beyond documentation and turn them into first-class citizens in software engineering, e.g., by deriving executable code from them or facilitate quality assessment [66].

MDE formalizes construction specifications for models as *modeling languages* [10]. A modeling language consists of (i) an abstract syntax defining available modeling concepts; (ii) one or more concrete syntaxes whose constructs allow concept instantiation by users; and (iii) semantics that assign meaning to concepts. MDE suggests the realization of *model processors* to elevate models from documentation to engineering artifacts. Examples of model processors comprise code generators, static analyzers, and interpreters [10].

Given its focus on abstraction, MDE is particularly applicable in the design, development, and operation of complex software systems [19], and their architectures [56]. Its adoption has benefited software engineering in heterogeneous domains such as cyber-physical systems [34], Industry 4.0 [67], Internet of Things [31], and Service-Oriented Architecture (SOA) [3].

B Model-Driven Microservice Engineering with LEMMA

The successful adoption of MDE for SOA (Appendix A) stimulated research on its application to MSA [23, 2, 63, 61, 28, 25]. LEMMA (Language Ecosystem for Modeling Microservice Architecture) [49] is an approach to *MDE-for-MSA* [21] that copes with MSA's complexity by concern-driven decomposition. To this end, it provides integrated modeling languages for the modeling of a microservice architecture from different *viewpoints* [24]. A viewpoint addresses selected stakeholder concerns, and prescribes notations and processing instructions for architecture models reifying such concerns. Figure 4a shows LEMMA's viewpoints, their stakeholders, modeling languages and model types, and Table 4 describes them.

LEMMA provides an import mechanism to let modeling languages prescribe reference relationships between modeling concepts. This mechanism supports model integration across viewpoints to increase models' information content. Figure 4b depicts the import relationships between LEMMA's model types. The following import relationships can be established:

- **Domain Model:** Domain models can import domain concepts from other domain models to use these external concepts as types for local concepts.
- **Service Model:** Service models can import microservices from other service models to capture service dependencies. Furthermore, service models can import domain concepts from domain models to use them as types of operation parameters. To configure a microservice to exploit a certain technology, service models can be augmented with information captured by imported technology models, e.g., protocols or aspects (Table 4).
- **Mapping Model:** Mapping models import microservices and, transitively, domain concepts from service models as well as technology information from technology models. They can then augment microservices and domain concepts with technology information.
- **Technology Model:** Technology models import other technology models to specify conversion directions between technology-specific types.

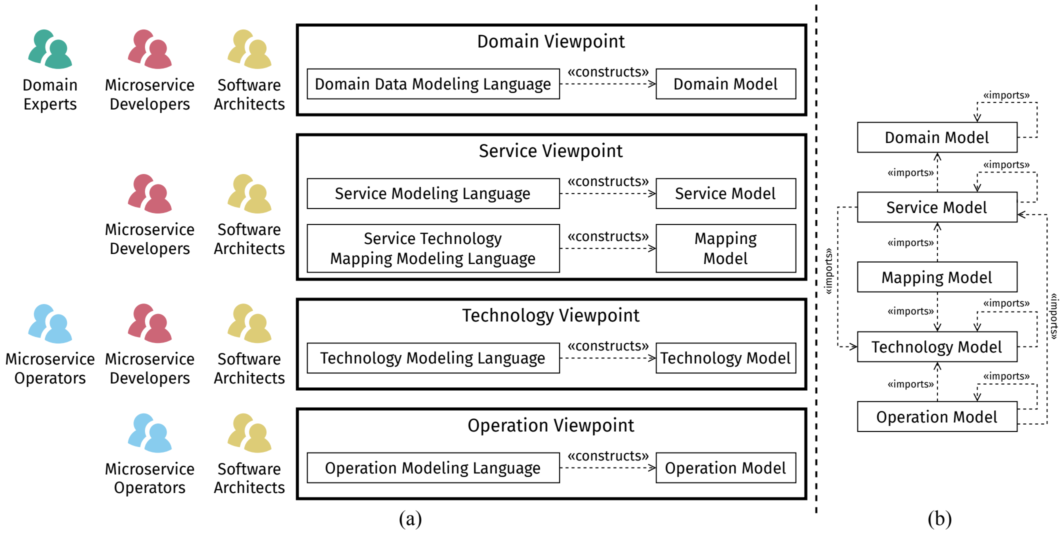


Figure 4 (a) LEMMA's viewpoints, their stakeholders, modeling languages and model types. (b) Import relationships between LEMMA model types.

- **Operation Model:** Operation models import other operation models to identify dependencies between operation nodes. Moreover, they can import technology and service models for node configuration and container-based microservice deployment, respectively.

Next to viewpoint-specific modeling languages, LEMMA also bundles its own MPF to facilitate the implementation of model processors by technology-savvy MSA stakeholders, e.g., microservice developers and operators (Table 4). The MPF applies the Phased Construction pattern [32] to structure model processing into the following phases:

1. **Model Parsing:** Parses input LEMMA models into object graphs to allow their efficient traversal. The phase is automated and does not require knowledge of MDE technologies.
2. **Model Validation:** Supports the implementation of validity checks as part of model processors. For instance, a code generator for event-based microservices might first ensure that modeled microservices exhibit operations that actually permit event handling.
3. **Code Generation:** Since code generation is among the key drivers for practical MDE adoption [66, 4, 33], LEMMA's MPF integrates a corresponding phase. However, since the MPF does not impose any requirements on generated code, the Code Generation phase may also be exploited to reify results from other model processing purposes, e.g., quality attribute values calculated during static model analysis [49].

LEMMA's MPF draws on popular MSA technologies and mechanisms such as the Java programming language [57, 7] and class-based Inversion of Control [26, 58]. Therefore, it provides specialized Java annotations for the Model Validation and Code Generation phases. The MPF detects these annotations at runtime on classes and methods, and handles annotated elements as intended by the phase, e.g., for signaling errors in validated models.

C Complexity Assessment for Extending LEMMA with Support for All Architectural Patterns in Sect. 2

Table 4 Description of LEMMA's viewpoints, stakeholders, modeling languages and model types.

Viewpoint	Stakeholders	Modeling Languages
Domain	Domain Experts, Microservice Developers, Software Architects	*Domain Data Modeling Language*: Enables to construct *domain models* with domain-specific data structures, collections, and enumerations via Domain-driven Design [15].
Service	Microservice Developers, Software Architects	*Service Modeling Language*: Supports the construction of *service models* for microservices, interfaces, and operations. *Service Technology Mapping Modeling Language*: Enables the construction of *mapping models* that assign technology-specific information to domain or service model elements. Models can therefore remain technology-agnostic and reusable across alternative technology choices [36].
Technology	Microservice Operators, Microservice Developers, Software Architects	*Technology Modeling Language*: Allows the construction of *technology models* that capture types of service programming languages, communication protocols, and operation technologies. In addition, generic *technology aspects* can be specified to augment elements in LEMMA models, e.g., data structures and microservices, with technology-specific information like database mappings or endpoint locations.
Operation	Microservice Operators, Software Architects	*Operation Modeling Language*: Constructs *operation models* for microservice deployment and infrastructure including its usage by services.

Table 5 Assessed complexity of extending LEMMA with support for the architectural MSA patterns described in Sect. 2.

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Complexity of Extension[b)]
		Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
Communication	**API Gateway**			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	API Composition	✓	✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for API composition.							
	Gateway Routing			✓	○	○	○	○
	Comment: Requires identification of API gateway as router.							
	Gateway Offloading		✓	✓	●	○	●	●
	Comment: Number of concepts likely increases with offloaded service functionality.							
	Event Sourcing	✓	✓	✓	○	◐	●	◐
	Comment: Requires persisting message broker component. Services interacting with broker must only send/receive domain events.							
	Log Aggregator			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Extension Complexity[b)]
		Dom.	**Serv.**	**Op.**	**Concepts**	**Constraints**	**LOC**[c)]	
Deployment	**Sidecar**		✓		◐	◐	●	◐
	Comment: Requires Sidecar identification and assignment to business-oriented service.							
	Microservice Chassis		✓		●	●	●	●
	Comment: By contrast to Sidecar: Requires delegating of all cross-cutting concerns.							
	Backend for Frontend	✓	✓	✓	○	○	●	◐
	Comment: Additional separate backends for highly diverse, client-specific frontends.							
	Database is the Service			✓	○	◐	○	○
	Comment: Requires validation that each service has its own database.							
Design	**Processing Resource**	✓	✓		○	○	●	N/A
	Comment: Fully supported by LEMMA's domain and service modeling concepts.							
	Parameter Tree	✓			◐	◐	○	◐
	Comment: Requires identification of domain concept containments.							
	API Key	✓	✓	✓	○	◐	●	●
	Comment: Requires identification of API key fields and mature security infrastructure.							
DevOps	**Externalized Configuration**		✓	✓	○	◐	◐	◐
	Comment: Requires configuration provider component to be used by services.							
	Monitor			✓	○	○	●	●
	Comment: Requires monitor component to be used by services.							
	Application Metrics			✓	○	○	●	●
	Comment: Requires metrics collector component to be used by services.							
	Distributed Tracing		✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for service interaction.							
	Health Check			✓	○	○	●	●
	Comment: Requires health checker component to be used by services.							
Migration	**Strangler**	✓	✓	✓	○	○	●	N/A
	Comment: Fully supported by LEMMA's modeling concepts for the viewpoints.							
	Anti-Corr. Layer	✓	✓		○	○	●	N/A
	Comment: Fully supported by LEMMA's domain and service modeling concepts.							
	Decompose by Business Cap.	✓			N/A	N/A	N/A	N/A
	Comment: Pre-MSA systems are out of LEMMA's scope.							

Category	Pattern	LEMMA Viewpoints			Quantities[a)]			Extension Complexity[b)]
		Dom.	Serv.	Op.	Concepts	Constraints	LOC[c)]	
	Decompose by Subdomain	✓			N/A	N/A	N/A	N/A
	Comment: Pre-MSA systems are out of LEMMA's scope.							
Orchestration	**Container**			✓	○	○	●	N/A
	Comment: Fully supported by LEMMA's Operation Modeling Language.							
	Serv. Registry			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Serv. Discovery			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Load Balancer			✓	○	○	○	○
	Comment: Requires only one infrastructure component.							
	Circuit Breaker		✓		○	○	◐	○
	Comment: Requires identification of circuit breaker capability on services.							
	Saga	✓	✓	✓	●	●	●	●
	Comment: No built-in modeling concepts for service interactions.							

Symbol key: ○ = Low; ◐ = Middle; ● = High.
a) Assessed relative across all patterns.
b) Assessed based on quantities, and experiences with DOMAIN EVENT (Sect. 3) and CQRS (Sect. 4).
c) Assessed quantity of generated code.

Applying QoS in FaaS Applications: A Software Product Line Approach

Pablo Serrano-Gutierrez
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain

Inmaculada Ayala
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain

Lidia Fuentes
Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain

Abstract

A FaaS system offers numerous advantages for the developer of microservices-based systems since they do not have to worry about the infrastructure that supports them or scaling and maintenance tasks. However, applying quality of service (QoS) policies in this kind of application is not easy. The high number of functions an application can have, and its various implementations introduce a high variability that requires a mechanism to decide which functions are more appropriate to achieve specific goals. We propose a Software Product Line based approach that uses feature models that model the application's tasks and operations, considering the family of services derived from the multiple functions that can perform a specific procedure. Through an optimisation process, the system obtains an optimal configuration that it will use to direct service requests to the most appropriate functions to meet specific QoS requirements.

2012 ACM Subject Classification Computer systems organization → Reconfigurable computing; Computer systems organization → Cloud computing

Keywords and phrases FaaS, Serverless, QoS, Software Product Line, Feature Model

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.9

Funding This work is supported by the European Union's H2020 research and innovation programme under grant agreement DAEMON 101017109, by the projects co-financed by FEDER funds LEIA UMA18-FEDERJA-15, IRIS PID2021-122812OB-I00 and by the project DISCO B1-2012_12 funded by Universidad de Málaga.

1 Introduction

Functions as a service (FaaS) are a type of service based on cloud computing, which allows the development of systems based on functions managed by the platform itself, freeing the developer from the tasks of scaling and maintenance. In a FaaS approach, Software systems are developed using a set of independent functions that perform the tasks required by the application. These are serverless [2] functions that may behave as microservices or nano-services but with the advantage of being fully managed.

Another great advantage of FaaS applications is the possibility of choosing different functions to perform the same task or operation without modifying the application. These functions can be implemented by independent teams or based on various algorithms. Thus, deciding which available function is more suitable is essential because function performance at runtime can differ depending on the operational context. It is necessary to consider both functional and non-functional requirements since different implementations of a function may be decisive in the result of the execution of the application. Therefore, function selection is an essential concern in FaaS applications.

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 9; pp. 9:1–9:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Although many frameworks are available to implement FaaS systems, they do not provide mechanisms to apply quality of service (QoS) policies beyond controlling certain function-level aspects, such as scaling. So the developer must decide which functions are optimal to achieve the results required by the application and how to compose them. This task is arduous and not readily adaptable to changes in QoS requirements. We use cost and response time in this work, but we can apply this approach to other QoS parameters, such as security or energy consumption.

Software Product Lines [10] is an approach for software development that focuses on developing a family of software systems using reusable assets. To define the common and variable elements of these families of software systems, a widely used approach is Feature Models [9]. A feature model contains an explicit representation of the configuration space using features. A feature model organizes features into a tree, including the corresponding tree and cross-tree constraints representing feature dependencies. In addition, it can consist of information about what features are optional or mandatory in a final system. A valid configuration is a particularization of the feature model that complies with the imposed constraints.

In this work, we will use feature models to model the composition of the tasks of a FaaS application. Our feature models include the alternative functions to perform the operations that made a task and their constraints. Using these models with the Z3 solver, we will obtain valid and optimal configurations of our FaaS application, taking into account QoS policies.

The main objectives of this work are:

1. *Manage FaaS application QoS at runtime.* The application could request the system to adjust the QoS parameters pursued at any time.
2. *Generate the optimal configurations dynamically or, at least, those that meet restrictions.* The system recalculates the configurations every time there is a change in the QoS objectives.
3. *Decouple the serverless implementation from a specific serverless framework.* The system is not integrated into any framework but communicates with it through REST requests. This type of interaction makes it possible to use our proposal with most existing frameworks. At the same time, this system does not require any extra learning from the developer, who can continue working similarly.

This paper is structured as follows: in Section 2 we present a brief introduction on feature modeling, Section 3 discuses some related work; Section 4 introduces our approach; Section 5 presents our case study on Reservation systems; Section 6 exposes the results to the experiments carried out; and Section 7 presents some conclusions to the paper.

2 Background on feature modeling

A feature model [9] is a hierarchical specification of systems through a set of features, which are elements or properties that may or may not be present in a concrete system. Features are organised in tree structures representing the dependencies between them, so selecting a leaf for the final system depends on selecting its ancestor features. In addition, it is possible to define explicit constraints, also known as cross-tree constraints. For example, if we consider a multi-functional system composed by a printer and a scanner (see Figure 1), a Printer with the feature *Photo* may not be compatible with *GreenMode*, in that case, we can add the cross-tree constraint *Photo excludes GreenMode*.

Features can be divided into Boolean features (e.g., *Printer*), numerical features (e.g., *resolution:int*), and variability sub-tree (e.g., *Cartridge[1..4]*). Boolean features represent yes/no decisions or features that can appear in the final system or not. Numerical features

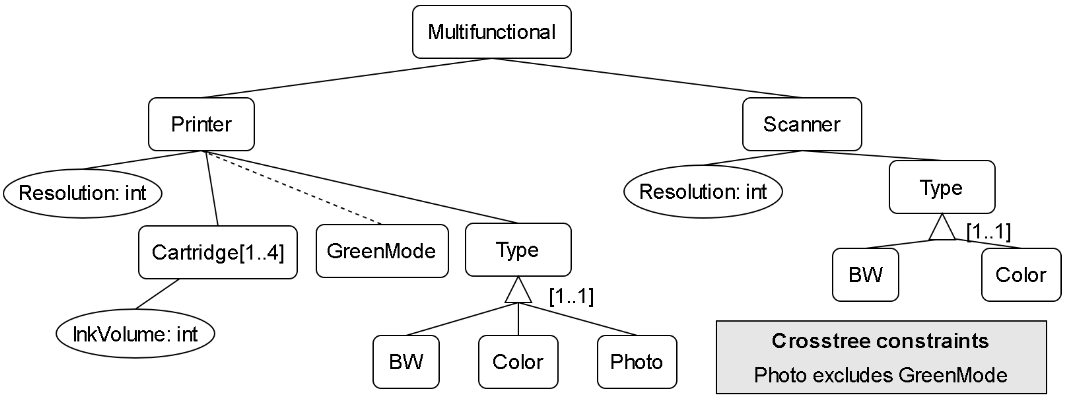


Figure 1 Feature Model generation.

are features that require a value to be resolved. Variability sub-trees mean creating instances or clones and providing per-instance resolution for features in its sub-tree. Cardinality features or feature relations (e.g., *Type*) apply restrictions over the number of children of a feature that can be chosen. The appearance of a feature in a selection can be mandatory, linked using solid lines, or optional, linked using dashed lines (e.g., *GreenMode*). A feature model or product configuration is a selection of features that respect the tree and cross-tree constraints defined in the feature model. A configuration that complies with explicit and implicit restrictions of a feature model is considered valid.

3 Related work

There are not many jobs that consider QoS parameters in FaaS environments. Some of them focus on the performance of individual functions due to the work of the framework itself [8]. Other articles, such as [7], consider the global performance of the application but, in this case, working at the FaaS platform level, controlling the location of resources. Also noteworthy is [11], in which Sheshadri K et al. present a system that allows specifying a QoS for a FaaS application and that has a QoS-aware resource manager that intervenes in deployment decisions. It also works on resource requirements, trying to use them efficiently. Our system uses a complementary approach to these, working on the composition of application functions so that any efficient resource allocation methods could be applied together.

On the other hand, numerous works that use Software Product Lines to model the variability of services can be found. In [12], the authors use feature models to recalculate service composition after a failure. Also, M. Abu-Matar et al. [1] model the variability of web services using Software Product Lines.

Likewise, numerous articles dealing with the selection of services consider QoS. One is [4], which shows how to compose web services considering different quality attributes. In microservices, [6] uses a model of the service workflow to select microservices utilizing an algorithm based on list scheduling. Nevertheless, unlike our proposal, they do not use a Software Product Line approach that models the variability due to multiple function implementations.

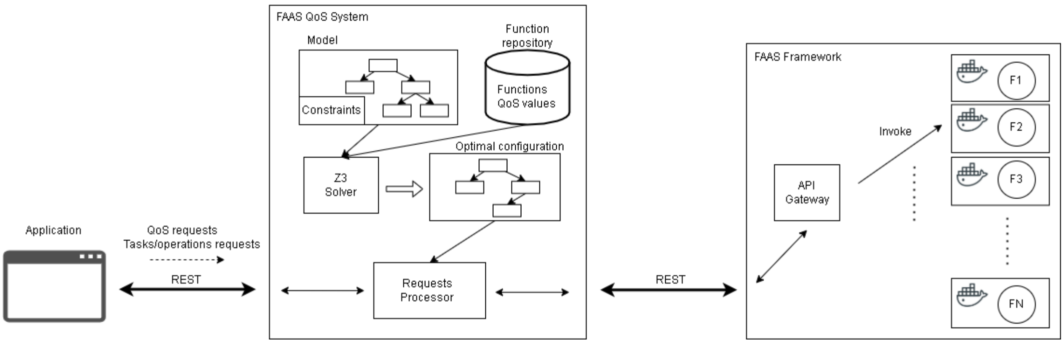


Figure 2 Proposed FaaS QoS system.

4 Our approach

Our solution consists of a system (see Figure 2) that, based on FaaS, allows us to choose at runtime the best functions that perform the operations needed by the application to offer a specific QoS. The proposal is based on the Software Product Line approach, using feature models to specify the variants of a particular service or task of a workflow. Our system performs this task as transparently as possible to the developer. So, there is no need to worry about knowing the functions' different implementations and characteristics.

Based on a feature model of the application, the proposed system calculates the optimal selection of the tasks and functions that is more convenient to meet specific QoS. This optimisation is carried out using the Z3 solver, which uses the model and the characteristics of each available function to carry out the different tasks necessary for the application.

4.1 The application model

Before optimising the feature set, our system needs to obtain a model of the application to work on. Initially, it is necessary to build a model of the application workflow. This is made up of a series of tasks that are modelled as features in a feature model. At the same time, each task is performed through a certain number of operations necessary to complete it. These operations are the ones that we will execute through FaaS functions in our system. Different implementations of the functions may carry out each of these operations. Each function has specific associated characteristics for each operation, such as execution times, costs, security, etc. As a second step, we use this information to automatically generate a collection of service feature models representing the variability of each executing task. In addition, our system takes into account another source of variability. Some functions may have parameters that vary their behaviour, sometimes affecting in some way the operation QoS. These parameters are specified as attributes associated with the corresponding feature. For example, a video playback function may operate with different resolution values and several possible frames per second. For this reason, in the third step, we will automatically generate function feature models, incorporating these new features derived from appropriate options that can modify the function's behaviour in terms of QoS.

Therefore, at this point, we have a *feature model of the application*, *a set of service feature models* related to the tasks, and *a set of function feature models* associated with each serverless function. These models constitute the global application model that our system will use.

4.2 Analysis system of the application model

Based on the feature models obtained, our system performs an analysis to generate an optimal configuration. To accomplish this task, we use a function repository that contains information about the QoS values of the different configurations of the FaaS functions, each implemented by a set of operations. Finally, an optimisation process can be carried out considering a set of restrictions related to the QoS to be achieved. A valid configuration that adjusts to the referred feature models is obtained through this process whenever possible. This configuration will be the one that allows the desired QoS to be achieved and will enable the system to choose the tasks and functions to be executed.

4.3 Valid Configuration Generation

With the models obtained in the steps described above, and applying Software Product Lines refactoring techniques [3], we generate a Feature Model with the information obtained from the entire application. To determine the valid configurations, we use a mathematical model built from the application's Feature Model and add existing and user-specified restrictions. The conversion of the features to this model is done automatically. The main component of this conversion is a recursive process which traverses a tree that represents the relationships contained in the feature models and obtains all the logical connections between features. When a node is reached, the set is assigned the AND, OR, or XOR logical relationship indicated in the parent node. The process ends with obtaining a logical expression that represents the entire tree. To denote it, the following expression will be used:

$$L(t_1, \dots t_{|T|}) \qquad t_i \in Bool \tag{1}$$

Where T is the set of tasks and t_i is a variable that identifies the task i, which is a Boolean value that indicates if the task is executed or not. L represents a function that relates these t_i through the logical expression that represents the task tree of the application, obtained through the previous procedure. Next, we must add the necessary equations to model the variability of the operations, taking into account that each can be carried out through a set of different functions. The following equation expresses this:

$$\sum_{i=1}^{n} f_{ijk} = 1 \qquad \forall k \in [1, |T|], \forall j \in [1, |O_k|], f_{ijk} \in \{0, 1\} \tag{2}$$

Where f_{ijk} is associated with the serverless function F_{ijk} that implements some of the operations used by some of the tasks that are part of the application. It can take a value of 0 or 1 that will indicate, respectively, if that function is part of the configuration or not. On the other hand, O_k is the set of operations that make up a given task t_k. Therefore, $\{f_{1jk}, \dots f_{njk}\}$ represents the set of n functions that can perform the same operation o_j related to the task t_k.

The equations (1) and (2) are used to identify a configuration. If, in addition, we add the constraints defined on the tasks or the functions, we will obtain a set of valid configurations. We will call CT this set of constraints, which will be represented as logical functions ct_i that relate tasks, functions and parameters, as defined in the following equation:

$$ct_i(T, F, P, V) \qquad \forall i \in [1, |CT|] \tag{3}$$

Where F is the set of functions, P is the set of function parameters, and V is the set of values that the different parameters can take.

4.4 Optimisation Process

We use the equations defined in the previous section and the model constraints to carry out the optimisation process. We add to these equations another set of equations to associate the parameters related to functions' QoS with those of the complete system. These equations differ depending on the parameter considered since execution time differs from cost or security. In some cases, it is a matter of adding all the functions' values. In some cases, we must consider the tasks performed; in others, the values greater or lesser must be considered, etc. Finally, the constraints specified by the user are added. To obtain a valid configuration, the system applies the optimisation function provided by Z3 using the complete set of equations obtained. This configuration is optimal according to the specified parameters and complies with the user restrictions.

Our system can optimise using one or several QoS parameters: latency, execution time, security, cost and energy. We can easily extend our approach for new parameters by generating a specific set of equations. Some functions may have different associated values for the same QoS attribute. In this scenario, it is necessary to consider the feature model to calculate the associated QoS values. The set of equations to optimise cost or energy are the following:

$$
\begin{aligned}
Min.: \quad & \sum_{k=1}^{|T|}\sum_{j=1}^{|O_k|}\sum_{i=1}^{n} t_k \cdot f_{ijk} \cdot c_{ijk} \\
s.t.: \quad & L(t_1, \dots t_{|T|}) \quad t_i \in Bool \\
& \sum_{i=1}^{n} f_{ijk} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|] \\
& f_{ijk} \in \{0, 1\} \\
& ct_i(T, F, P, V) \quad \forall i \in [1, |CT|] \\
& \sum_{m=1}^{nv} fp_{jklm} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|], \\
& \qquad \forall l \in [1, |P_{jk}|] \\
& fp_{jklm} \in \{0, 1\} \\
& pv_{jkl} = Param(p_{jkl}, m) \forall p_{jkl} \in P_{jk}, \forall fp_{jklm} = 1 \\
& c_{ijk} = Cost(F_{ijk}, \{pv_{jk1}, \dots pv_{jkn}\}) \forall F_{ijk} \in F
\end{aligned}
\tag{4}
$$

Where P_{jk} represents the set of l parameters of the operation j related to task k, and fp_{jklm} is a variable that indicates the value assigned to the parameter p_{jkl} from a set of nv possible values, it can take values 1 or 0, meaning that the value m is selected or not, respectively. For example, considering a rendering function whose result depends on the output resolution, which can take three possible values, such as *480px*, *640px* and *800px*, and that *Render* is the first operation of the second task, fp_{1212} indicates whether the value *640px* is chosen or not for its first parameter *Resolution*.

Cost is a function that returns the cost of the function F_{ijk} considering a set of values $\{pv_{jk1}, \dots pv_{jkn}\}$ assigned to its parameters. *Param(p,i)* is a function that returns the value for the parameter p that is determined by the index i from the list of possible values for this parameter. So, following with the last example, $Param(P_{121},1)=640px$, and *Cost(RenderA,640px,1)* will return the cost of the operation *Cost(Render)* with *640px* as the value for the input parameter, if the function *RenderA* is used to perform that operation. As mentioned before, this could be equally used for energy. In that case, the function *Cost* would be Energy and would return the Energy consumed by the execution of the function F_{ijk}.

If we want to optimise execution time or latency, the considered equations are:

$$\begin{aligned}
Min.: \quad & max(\{s|s = t_k \cdot f_{ijk} \cdot s_{ijk} \forall k \in [1, |T|], \\
& \quad \forall j \in [1, |O_k|], \forall i \in [1, |F_j|]\}) \\
s.t.: \quad & L(t_1, \dots t_{|T|}) \quad t_i \in Bool \\
& \sum_{i=1}^{n} f_{ijk} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|] \\
& f_{ijk} \in \{0, 1\} \\
& ct_i(T, F, P, V) \quad \forall i \in [1, |CT|] \\
& \sum_{m=1}^{nv} fp_{jklm} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|], \\
& \quad \forall l \in [1, |P_{jk}|] \\
& fp_{jklm} \in \{0, 1\} \\
& pv_{jkl} = Param(p_{jkl}, m) \forall p_{jkl} \in P_{jk}, \forall fp_{jklm} = 1 \\
& t_{ijk} = Time(F_{ijk}, \{pv_{jk1}, \dots pv_{jkn}\}) \forall f_{ijk} \in F
\end{aligned} \tag{5}$$

Where *Time* is a function that returns the execution time of an operation performed by a determinate function. Since these values are used comparatively, the best solution is reached when mean values are considered. In the same way as before, we can substitute the *Time* function for *Latency* to optimise this parameter.

On the other hand, if we want to optimise some measurable aspect related to security, we will have:

$$\begin{aligned}
Max.: \quad & min(\{s|s = t_k \cdot f_{ijk} \cdot s_{ijk} \forall k \in [1, |T|], \\
& \quad \forall j \in [1, |O_k|], \forall i \in [1, |F_j|]\}) \\
s.t.: \quad & L(t_1, \dots t_{|T|}) \quad t_i \in Bool \\
& \sum_{i=1}^{n} f_{ijk} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|] \\
& f_{ijk} \in \{0, 1\} \\
& ct_i(T, F, P, V) \quad \forall i \in [1, |CT|] \\
& \sum_{m=1}^{nv} fp_{jklm} = 1 \quad \forall k \in [1, |T|], \forall j \in [1, |O_k|], \\
& \quad \forall l \in [1, |P_{jk}|] \\
& fp_{jklm} \in \{0, 1\} \\
& pv_{jkl} = Param(p_{jkl}, m) \forall p_{jkl} \in P_{jk}, \forall fp_{jklm} = 1 \\
& s_{ijk} = Secur(F_{ijk}, \{pv_{jk1}, \dots pv_{jkn}\}) \forall f_{ijk} \in F
\end{aligned} \tag{6}$$

Where *Secur* is a function that returns the security level of the aspect considered, provided by a serverless function. Each implementation is assigned a value that will be higher the more secure the function is considered. For example, if we have a function to login and it does not use SSL, the security level could be 1, which would correspond to a low level; in case of using it, we could assign a value of 2, and if, in addition, it uses two-step verification, a value of 3 could be considered, corresponding to a high level of security. In this case, if the system uses an insecure function, the whole system became insecure, so in this optimisation process we will try to use the highest secure level of the aspect considered, for all the functions.

4.5 Function selection

As stated in the introduction, a FaaS application is built based on numerous independent functions handled by a framework. These serverless functions are deployed in containers and called by the application when needed, i.e., they behave as self-managed stateless microservices. These functions, being elements external to the application, can be altered without intervening in its functional behaviour. Therefore, it is possible to change the implementation of a function without modifying or rebuilding the application. So, it is common to have several function implementations that can perform the same operation.

We have implemented our function selection component using REST requests to use this system with different serverless frameworks. REST makes possible a natural integration of this component in FaaS platforms because they are commonly used to call serverless functions. Specifically, these frameworks use a service called API Gateway. Using this service, they receive REST requests of functions from the FaaS application and send them to the implementations of the corresponding functions deployed in containers in the cloud. Our system, programmed in Python, attends to calls made like they would be made to a FaaS framework.

The system receives a generic request and transforms it into a specific request for a function managed by the FaasS framework. For example, suppose we want to perform an operation that compresses an image. In that case, we use a generic name like *Compress* in the code instead of calling a specific function that compresses the image using a particular algorithm. Our system will transform this request into the corresponding call to the FaaS framework so that a compression function implemented using a specific algorithm is executed.

4.6 Requests processing

The system receives the requests from the application, both for functions and adjustment of QoS parameters. When the request is for the execution of a function, in the first step, it processes, if necessary, the input parameters to the function following the configuration obtained by the analysis module and reconstructs the request. The application can avoid assigning these optimal parameters to the function by passing the desired values in the request. In the next step, the processed request is passed directly to the function selector, which will redirect the request to the appropriate implementation deployed by the FaaS framework. On the other hand, the platform admits the entry of requests related to the QoS to which it is desired to adjust the operation of the application. To distinguish feature requests from QoS adjustment requests, the system listens on two different ports that can be previously configured. These requests also follow the REST style and are confirmed in the same way. Once the function is executed, the response of the function is passed directly to the application in response to the execution request. This also occurs in the form of a REST request.

We have considered two possible modes of operation to meet the needs of different types of applications. On the one hand, it is possible to work entirely transparently. In this case, the application that uses our platform must only follow the process described, and it makes QoS requests when needed, and every time it has to perform an operation, it makes a request for a said operation to the system. The developer, therefore, programs the application in the same way as if we were working directly on a FaaS framework. This way, it would also apply to already-built FaaS applications.

On the other hand, it is possible to work interactively. In this mode, the application can query the system for information on the optimal configuration. The operation is the same as we have described, with the addition of requests related to some of the tasks or operations.

Thus, if the application asks for the task T, it will be able to know if it is included in the optimal configuration calculated by the analysis module. In this case, the application does not work like a traditional FaaS application since it must interact with the platform.

As part of our approach, a repository contains the list of functions the application can use to perform the operations performed in a task. Each time a new implementation is made for one of these functions, it is only required to add the corresponding information to the repository. Then, this information is automatically integrated into the generated models.

A traditional FaaS application makes function calls through a framework used to implement FaaS. These frameworks use an API Gateway element, through which they receive requests, sending them to the corresponding functions, which are placed in containers in the cloud. The system delivers the execution results to the calling application through this same Gateway. Of course, these systems consist of other elements, such as an orchestrator responsible for deploying and maintaining the containers. Communication with the API Gateway is often done through REST requests, which is why it has also been chosen as the communication mechanism for our system so that it is as similar as possible to working directly with the FaaS framework. The difference is we work with tasks and operations instead of specific functions. So, suppose we want to perform an operation that compresses an image. In that case, we will use a generic name such as *Compress* instead of calling a function that performs the compression using a specific algorithm. Our system will be the one that makes the appropriate call to the FaaS framework so that a compressing function implemented through a particular algorithm is executed.

5 Case Study

To illustrate our proposal, we use an application to make travel bookings. The system considers that a booking is made of three different elements. Firstly, the hotel reservation, which will be regarded as mandatory for all travel bookings, and, on the other hand, the two possible transportation reservations (that are optional), which may be flight or train reservations. If there is a transportation reservation, the application could choose either of the two to complete the trip reservation. Therefore, in BMPN 2.0 notation, the application workflow (see Figure 3), which represents its functional requirements, will have two gateways. A first gateway indicates that the booking application can perform hotel and transport reservation tasks in parallel. A second gateway shows the two possible alternatives in the transport branch, as shown in Figure 3. The correspondence of this workflow with the associated feature model is trivial. Each task is represented as a feature of the feature model. The parallel, exclusive and inclusive gateways will be modelled as feature relationships of type AND, XOR, and OR, respectively. Thus, AND will represent feature groups with two or more obligatory children, XOR will encompass feature groups with exclusive alternative children, and OR will be feature groups with alternative children. For example, for the tasks in our case study, we would get: *Hotel_Book AND (Flight_Book XOR Train_Book)*. As we can see, the system is easily adaptable to the characteristics of the application. For example, if we consider an alternative to the hotel for the accommodation, such as an apartment. Then, it would be enough to modify the model indicating that *Hotel Book* is not mandatory but optional and integrate it in an XOR branch together with the new *Apartment* Book task.

Each of these three tasks is performed by carrying out a series of operations, as seen in the task diagrams in Figure 4. In the case of hotel reservations, the best available price for a room is obtained, and subsequently, the reservation is made. Likewise, *Train Book* and *Flight Book* must carry out a series of operations to complete the train and flight reservation,

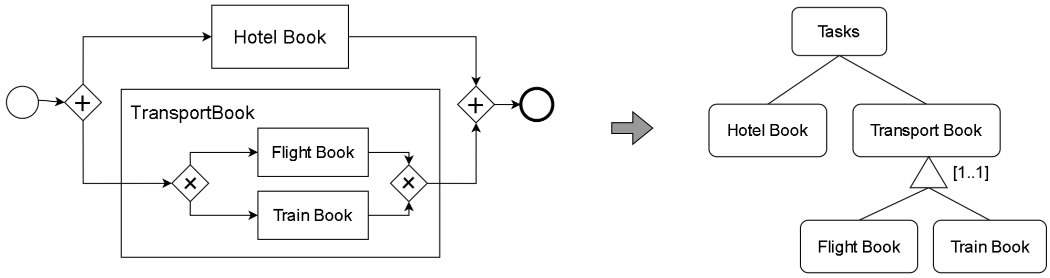


Figure 3 Workflow of the application and Tasks Feature model.

respectively. Each of these operations is performed by a FaaS function corresponding to a function listed in the repository. There are alternative functions to perform each operation, and they are labelled with cost and execution time values. In order to compare results, we have measured these values in the same conditions and stored them in the repository. As can be seen in this example, the system can support the use of additional factors to perform the optimisation, since it is enough to include the values associated with each function in the repository. Thus, in this case, we not only take into account the performance of the functions but we add the cost of the reservation as a factor to take into account. For example, in the case of the *Booking* operation, we find three alternatives, *BookingA*, *BookingB* and *BookingC*, in which option *BookingB* is the fastest but at the same time has the highest cost. However, *BookingA* is slower but has the lowest price. We can model additional restrictions, for example, *BookingC implies TicketPurchaseA*, which means that if the reservation is made with the *BookingC* function, it is necessary to purchase the train ticket with the function *TicketPurchaseA*. In a real case, it can correspond to the condition of making two payments using the same banking service to obtain lower surcharges. For our case study, these constraints are those shown in the upper box of Figure 4.

With the available tasks and functions information, we model the feature models that appear in the central box of Figure 4. These feature models represent all the variability introduced by the tasks and the different implementations of the functions. These feature models are the primary input of an optimisation process that finds the best configuration of the FaaS application meeting restrictions related to the QoS. In the example, the aim is to minimise the response time while restricting the cost, imposing that it must be less than 10. It is necessary to consider that the response time will correspond to that of the slowest branch of the tree, while the cost refers to the set of all tasks performed. The Z3 solver works with the feature model and the above mentioned restrictions to obtain a valid configuration of functions that achieves the desired QoS. In this case, we get a combination of hotel and train reservations. Despite being the fastest, we can see how the *BookingB* option has not been chosen. In this case, it is because the cost of the system would then exceed ten units in our example.

Using this calculated configuration, our system is ready to receive requests from the application and process them according to it. Thus, when it gets a *FindRoom* request, it will proceed to call the *FindRoomA* function through a request to the FaaS framework and return its result to the application.

To illustrate the difference between the two modes of operation (see Figure 5) described in section 4.6, we can think of a trip reservation application in which a hotel and transport, which can be a train or plane, must be reserved, the objective of which is to complete a reservation with the lowest possible cost. If the user chooses hotel and train, the system will

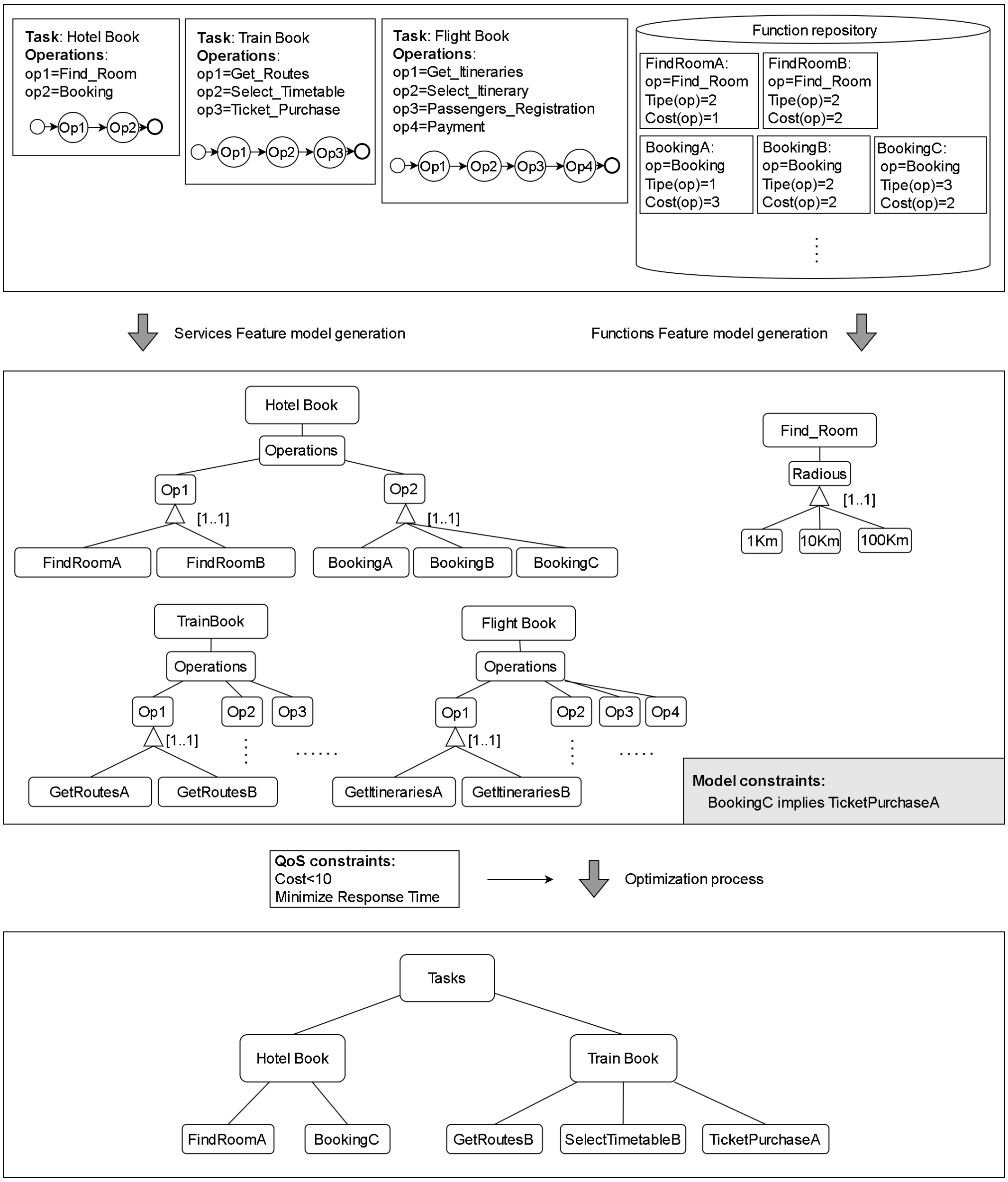


Figure 4 Feature model and configuration generation.

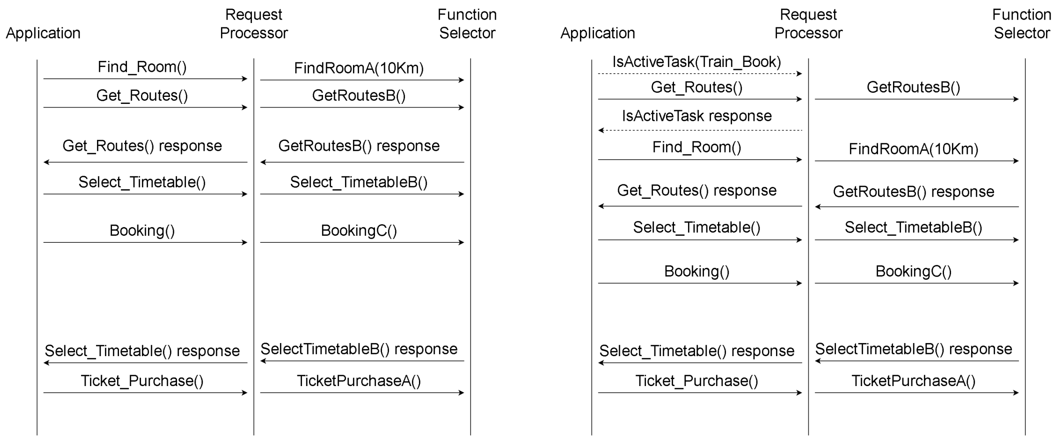


Figure 5 Sequence diagrams of the interaction with the request processor in transparent mode (left) and interactive mode (right).

generate an optimal configuration for that combination. However, we can also consider a different behaviour where the user does not select anything. In that case, the system finds the best alternative to reduce the cost as much as possible: hotel and train or hotel and plane combinations. Then, the application must know which of the two transport-related tasks it should perform, the interactive mode described above is then necessary.

6 Experimental results

A series of experiments have been carried out to evaluate the performance of our platform. In addition to the case study, larger workflows and many implemented functions have been considered. The system is programmed in Python v.3.10.4 with Z3 v.4.8.15, and the hardware used is a PC Intel i5-7400, 3.00 GHz, 24 GiB of RAM.

6.1 Case study

Our case study comprises three tasks with two, three and four operations. The number of functions considered is 20, that is, between two and three for each operation. We have carried out 10,000 executions to obtain a reliable average value of the different time measurements. The results show the system is fast for this simple case, getting times less than two milliseconds.

We have considered three aspects to evaluate the performance of the designed system. Firstly, the processing of information on the different QoS aspects related to the functions extracted from the information in the functions repository. Secondly, the generation of the feature model from the workflow and the QoS data calculated in the previous step. And finally, the duration of the optimisation process carried out by Z3. Running our case study resulted in a mean total time of 1.89 milliseconds, a mean QoS data processing time of 0.91 milliseconds, a mean modelling time of 0.45 milliseconds, and a mean optimisation time of only 0.47 milliseconds.

6.2 General performance and scalability

To carry out a study of scalability, it is necessary to ask first what are the acceptable values for the parameters that influence the model's variability. This is important because the five levels that can be present in our system mean that a slight increment in these parameters supposes a substantial increment in the total number of possible configurations, which can lead to an exponential growth in complexity.

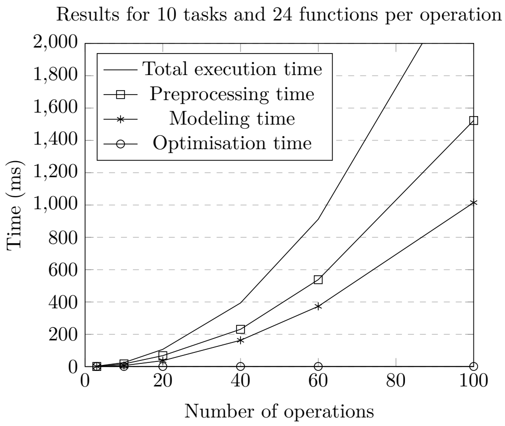
The first level to consider is the number of tasks. However, this will usually not be a very high number, usually less than ten [5] due to the high granularity of these systems, which means that the weight falls mainly on the number of operations performed by the tasks since they are the ones associated with the functions that are executed. At the same time, each operation can be performed by a different function implementation. Also, if a function can admit other parameters, they must be considered. From the point of view of analysis, to prepare a study that does not depend on so many variables and, therefore, can be more easily represented, we will include the effect of the parameters together with the multiplicity of functions. If an operation admits two configurable parameters and each of them can have three possible values, an effect similar to that of multiplying the variability by a factor of 6 can be considered, so if, for example, we have four possible implementations per operation, it would be somewhat comparable to having 24 implementations. This allows us to compare the effects of the diversity of implementations with those of the diversity of parameters and values supporting such functions in the same graph. Therefore, we have included this number when making the measurements, even though we will not usually find such a large number of implementations that perform the same operation. We must consider that the number of implementations of the same function will not usually be significant. An average of 3 could be a reasonable value, considering that not all the operations will have different implementations of functions to perform them.

The tests are performed considering the worst case. It occurs when there are no defined user restrictions or constraints. This is because constraints reduce the variability tree (i.e., the number of possible configurations of the variability model), and complexity decreases. For this test, a value of 100 executions has been chosen to obtain the average values.

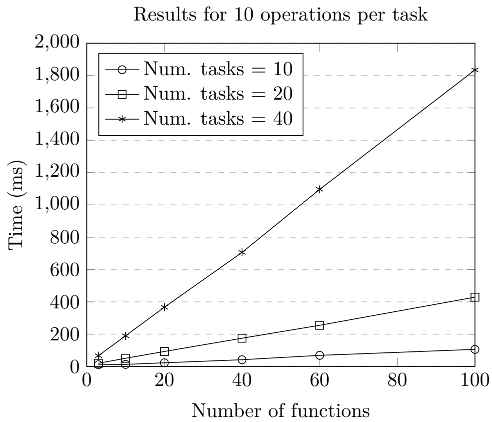
As we can observe in the graphic in Figure 6a, total execution times remain low even considering unrealistic values of the parameters that have been adjusted. Although it is observed that the growth of the times is exponential, even in a case with ten tasks, 100 operations per task and 24 functions per operation, the measured time barely exceeds two seconds. For the most common cases, the times are reduced to tens of milliseconds, which is quite a good performance.

Observing each of the times of the different processes that are carried out, we see that the behaviour is similar except in the case of optimisation, in which very similar values of a few milliseconds are obtained. Therefore, the preprocessing and modelling processes are the most time-consuming, with preprocessing slightly above. However, as mentioned, the total times are very short for real cases.

On the other hand, tests have been conducted considering ten operations per task to appreciate the effect of varying the number of functions and tasks. As shown in Figure 6b, the behaviour is practically linear when the number of functions is in this range. If we increase the number of tasks, total times also increase exponentially. However, even with 40 tasks, results remain contained and are lower than two seconds. Again, we see how the measured times are a few tens of milliseconds for more realistic values.



(a) Processing times varying the number of operations.



(b) Processing time varying the number of functions and tasks of the workflow.

Figure 6 Scalability tests results.

7 Conclusions

In this paper, we have presented a system that allows applying QoS parameters to a FaaS application. The proposed system uses feature models to model the variability of systems with multiple implementations of functions that can be chosen to perform a specific operation. Our proposal selects the best available functions to achieve a QoS, fulfilling a set of restrictions the user imposes. This will enable developers to make generic function requests, avoiding the need to know each of the implementations and their performance during coding. Thanks to a function repository, we can introduce new implementations of a particular function at runtime. Changing QoS requirements on the fly is also possible by automatically generating a new selection of functions. We plan to use this system to perform self-adaptive tasks based on changing conditions and consider the influence of other parameters external to the application, such as the infrastructure on which functions are deployed, that can also significantly affect QoS.

References

1 Mohammad Abu-Matar and Hassan Gomaa. Variability modeling for service oriented product line architectures. In *2011 15th International Software Product Line Conference*, pages 110–119, 2011. `doi:10.1109/SPLC.2011.26`.

2 Sarah Allen, Chris Aniszczyk, Chad Arimura, et al. Cncf serverless whitepaper, 2018.

3 Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1173706.1173737`.

4 Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007. `doi:10.1109/TSE.2007.1011`.

5 A. Dietzsch. Ratios to support the exploration of business process models in business process management, 2003.

6 Zhijun Ding, Sheng Wang, and Meiqin Pan. Qos-constrained service selection for networked microservices. *IEEE Access*, 8:39285–39299, 2020. `doi:10.1109/ACCESS.2020.2974188`.

7 MohammadReza HoseinyFarahabady, Young Choon Lee, Albert Y. Zomaya, and Zahir Tari. A qos-aware resource allocation controller for function as a service (faas) platform. In Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol, editors, *Service-Oriented Computing*, pages 241–255, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-69035-3_17.

8 Anisha Kumari, B. Sahoo, Ranjan Kumar Behera, Sanjay Misra, and Mayank Mohan Sharma. Evaluation of integrated frameworks for optimizing qos in serverless computing. In *ICCSA*, 2021. doi:10.1007/978-3-030-87007-2_20.

9 Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, pages 62–77, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-46020-9_5.

10 Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Berlin, Heidelberg, jan 2005. doi:10.1007/3-540-28901-1.

11 Sheshadri K R and J Lakshmi. Qos aware faas platform. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 812–819, 2021. doi:10.1109/CCGRID51090.2021.00099.

12 Jules White, Harrison Strowd, and Douglas Schmidt. Creating self-healing service compositions with feature models and microrebooting. *Int. J. Business Process Integration and Management Int. J. Business Process Integration and Management*, 1:0–0, jan 2009. doi:10.1504/IJBPIM.2009.026984.