FACULTY OF ENGINEERING AND ARCHITECTURE

Collaborative Compositions: Facilitating Service Orchestration from Cloud to Edge

Merlijn Sebrechts

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Information Engineering Technology

Supervisors
Prof. Filip De Turck, PhD - Prof. Bruno Volckaert, PhD
Department of Information Technology
Faculty of Engineering and Architecture, Ghent University

November 2022

GHENT UNIVERSITY

ISBN 978-94-6355-652-1
NUR 982, 986
Wettelijk depot: D/2022/10.500/93

Members of the Examination Board

Chair

Prof. Em. Hendrik Van Landeghem, PhD, Ghent University

Other members entitled to vote

Prof. Mays Al-Naday, PhD, University of Essex, United Kingdom
Bert Lagaisse, PhD, KU Leuven
Prof. Veerle Ongenae, PhD, Ghent University
Konstantinos Tsakalozos, PhD, Canonical, United Kingdom
Gregory Van Seghbroeck, PhD, Ghent University

Supervisors

Prof. Filip De Turck, PhD, Ghent University
Prof. Bruno Volckaert, PhD, Ghent University

Preface

"I suppose it's like the ticking crocodile, isn't it?
Time is chasing after all of us"
- J.M. Barrie, Peter Pan

This book is the culmination of the past seven years of my life. Seven incredible years that have changed me forever. It was both an intellectual and a personal journey that has touched my mind and heart forever. I took "the long route" by becoming a teaching assistant and taking some time off to become a cyborg, but it was completely worth it: I thoroughly enjoyed combining research with teaching, and I can't complain about being alive.

"If I have seen further, it is by standing on the shoulders of Giants"
- Isaac Newton, 1675

Collaboration has been an important theme in my research, but it has also been the sole reason why I was able to get where I am today. As such, I would like to use this soap box to thank all those who supported me along this journey. First and foremost, I would like to thank my supervisors Filip De Turck and Bruno Volckaert. Filip, thank you for giving me this opportunity to dedicate seven additional years of my life to education. I am still amazed by the amount of freedom you have given me to follow my interests in research and to hone my skills as an educator. I'm also thankful for your valuable feedback and your help navigating the more precarious parts of the academic world such as handling inappropriate reviewer comments. Bruno, thank you for the incredible amount of trust and support you gave me. It was incredible to work with you and it's inspiring to see someone so dedicated to both amazing research and high quality education. Although I was initially hesitant about whether I would be able to fit into the academic world, hearing about your story quickly put that to rest. I also want to thank Piet Demeester, director of IDLab, for creating such a professional and respectful environment that is a delight to work in.

I want to thank Gregory Van Seghbroeck for initially proposing the idea of doing a PhD and guiding my research during the early days. Our many late-afternoon brainstorms always left me with an incredibly amount of motivation and hundreds of new ideas for my research. Similarly, Thomas Vanhove was there from the beginning to help me think about the more practical applications of my research and to help me develop my professional communication. I also want to thank my

co-authors Mays AL-Naday for the intense discussions resulting in an incredible article, and Tim Wauters for his valuable feedback on my work. I also want to thank Tom Goethals for the many fruitful collaborations. Fun fact: three out of four papers that we authored together received a "best paper" award.

The open source software and communities I worked with over the past seven years have been invaluable. I still have fond memories of the Ubuntu Juju community. There are too many to mention, but I specifically want to thank Cory Johns, Alex Kavanagh and Stuart Bishop for letting me join the charms.reactive core team and for the many long discussions about improving code sharing between charmers. Fun fact #2: Long before my PhD, Ubuntu was my first introduction to Linux. Jorge O. Castro's infinite wisdom on AskUbuntu was invaluable to help me get started, so it was mind-blowing to be able to work with him on Juju, years later.

Special thanks goes to my coworkers for the incredibly stimulating lunch conversations about life, the universe and everything. I still fondly look back at the discussions with people like Leandro Ordonez, Stefano Petrangeli, Thijs Walcarius, Jerico Moeyersons, Laurens Van Hoye, Vincent Bracke, Maxim Claeys, Jeroen Van der Hooft and Femke De Backere. Also thanks to Wim Van de Meerssche specifically, for teaching me a lot about Linux, fielding endless questions about the intricacies of C and C++, and for his thoughtful and kind conversations. I also can't thank Sander Borny enough for the years of fruitful collaborations both in teaching and in research.

I was blessed to have had amazing support from friends during this period. Anne Fonteyn deserves an enormous amount of praise for helping me manage my stress during these final months of my PhD. Additionally, she went through great lengths to understand my work in order to help brainstorm the structure, title, and cover of this dissertation. Special thanks to Lies Warlop, for single-handedly pulling me through my master's and Ewaut Van Wassenhove for his unconditional support throughout the years. Thanks to Philip Vanloo and Rafael Mindreau to keep me sane throughout the COVID-19 pandemic, and thanks to all the "kiekens", who made sure I actually got away from my computer and socialized every once in a while.

I also want to thank my gigantic family for their support as well. Heidi, Pieternel, Simon, Mechtild, Pepijn and Amos Sebrechts; thank you for the many great family gatherings and inspiring discussions. Of course, I couldn't have done it without the support of my father, Hans Sebrechts and my mother Marianne Verstichel. Thank you for inspiring me to follow my passions and do great work, and thank you for cheering me on along the way.

Finally, thanks to my cats Simba and Panda. Although your contributions to my research were limited to drooling on my drafts, your truly unconditional love lifted me up every single day since you entered my life seven years ago.

Ghent, August 2022
Merlijn Sebrechts

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

AIOps	Artificial Intelligence for IT Operations
API	Application Programming Interface
AWS	Amazon Web Services

B

BPMN	Business Process Modeling Notation

C

CA	Certificate Authority
CD	Continuous Deployment
CDK	Canonical Distribution of Kubernetes
CI/CD	Continuous Integration and Continuous Deployment
CPU	Central Processing Unit

D

DAG	Direct Acyclic Graph

E

EC2	Elastic Compute Cloud

F

FSM	Finite State Machine
FWO	Flemish Fund Scientific Research (Fonds Wetenschappelijk Onderzoek - Vlaanderen)

G

GUI	Graphical User Interface

H

H&S	Hub and Spoke
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure

I

ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IQR	Interquartile Range
ISP	Internet Service Provider
ISV	Independent Software Vendors
IT	Information Technology
ITIL	Information Technology Infrastructure Library

J

JSON	JavaScript Object Notation

K

k8s	Kubernetes
Km	Kilometers

M

M@RT	Models@run.time
Mbps	Megabits per second

N

NFV	Network Function Virtualization

O

OA	Orchestration Agent
OS	Operating System

P

PaaS	Platform as a Service

Q

QoS	Quality of Service

R

RAM	Random Access Memory
REST	Representational State Transfer
RQ	Research Question

S

SA	Service Agent
SBO	Strategic Basic Research (Strategisch BasisOnderzoek)
SDK	Software Development Kit
SLA	Service-Level Agreement
SSE	Server-Sent Events
SSL	Secure Sockets Layer
STEM	Science, Technology, Engineering and Mathematics
sysadmins	System Administrators

T

TCP	Transmission Control Protocol
TOSCA	OASIS Topology and Orchestration Specification for Cloud Applications

U

UAV	Unmanned Aerial Vehicle
UI	User Interface

V

vCPU	Virtual Central Processing Unit
VLAIO	Flemish Agency for Innovation and Entrepreneurship (Vlaams Agentschap Innoveren en Ondernemen)
VM	Virtual Machine
VNF	Virtual Network Function
VPN	Virtual Private Network
VRT	Flemish Radio and Television (Vlaamse Radio en Televisie)

Y

YAML	Yet Another Markup Language

Samenvatting
– Summary in Dutch –

Samenwerking overheen organisatorische grenzen wordt steeds belangrijker. Dit gaat zowel over samenwerking tussen teams onderling, als tussen organisaties en bedrijven. De vierde industriële revolutie speelt hier een grote rol in doordat deze in gang gezet wordt door het verbinden van digitale systemen die de fysieke wereld manipuleren. In een industrie 4.0 bedrijf, bijvoorbeeld, zijn de machines van een productielijn onderling verbonden zodat ze informatie kunnen delen en hun gedrag aanpassen op basis van hun omgeving. Dergelijke productielijn wordt ook geconnecteerd met andere delen van een bedrijf om automatisch te rapporteren en met de bedrijfssoftware te integreren. Als laatste gaat de vierde industriële revolutie ook over het verbinden van de IT systemen van verschillende bedrijven om zo beter te kunnen inspelen op elkaars noden.

De complexiteit van IT systemen neemt toe door de vergaande digitalisering van de maatschappij. Softwaretoepassingen bestaan uit meer en meer componenten die samenwerken om een dienst te verlenen. Deze complexiteit maakt het lastig om applicaties stabiel te laten draaien en eenvoudig uit te breiden. Microservice architectuur probeert hierop een antwoord te bieden. Het is een manier om applicaties te ontwikkelen waarbij één applicatie achterliggend bestaat uit een hoop kleine componenten die elk door een apart team ontwikkeld kunnen worden. Het nadeel aan microservice architectuur is dat deze de complexiteit zelf niet oplost. Meer nog, deze manier van werken voegt zelfs nog meer complexiteit toe, bijvoorbeeld door de verhoogde communicatie tussen verschillende componenten van een applicatie. Hierdoor wordt het steeds moeilijker om grote applicaties te installeren en te beheren. Er is dus een hoge nood aan oplossingen om dit proces eenvoudiger te maken. Deze thesis pakt vijf specifieke moeilijkheden aan omtrent het beheren van applicaties die samengesteld zijn uit verschillende componenten die samenwerken.

De eerste moeilijkheid gaat over het vastleggen en hergebruiken van de kennis die een systeembeheerder heeft over welke componenten moeten opgezet worden voor het oplossen van een bepaald probleem. Huidige technieken om applicaties te beheren hebben het moeilijk om kennis van een systeembeheerder te encapsuleren. Deze zijn namelijk gebaseerd op declaratieve modellen die het gewenste eindresultaat heel uitgebreid beschrijven. Echter, indien de systeembeheerder niet weet wat er precies nodig is, dan kunnen deze tools niet helpen. Het is dus niet mogelijk om een model te ontwikkelen dat zelf beslist hoe de applicatie er moet uitzien. Hoofdstuk 2 van deze thesis stelt de “orchestrator conversation” voor. Dit is een framework waarmee systeembeheerders software kunnen ontwikkelen die zelf beslist wat er moet opgezet worden om bepaalde functionaliteit te voorzien. Het stelt het idee van een “orchestration agent” voor. Deze vertaalt abstracte modellen die op een hoog niveau beschrijven wat er nodig is, naar meer

concrete modellen die in detail beschrijven wat er moet opgezet worden. Dit hoofdstuk sluit af met een evaluatie van een prototype dat aantoont dat deze vertalingen gebeuren met minimale overhead.

De tweede moeilijkheid ligt in het vastleggen en hergebruiken van de kennis die een systeembeheerder heeft over wanneer welke beheersacties moeten uitgevoerd worden. Declaratieve modellen kunnen enkel de gewenste staat van een applicatie beschrijven. Echter, draaiende applicaties hebben ook onderhoud en updates nodig. Daarom bieden veel beheerstools de mogelijkheid aan om ook verschillende beheersacties te definiëren. De beheerstool beslist echter wanneer iedere actie wordt uitgevoerd op basis van een interne levenscyclus. Dit zorgt voor problemen omdat deze levenscyclussen zelden overeen komen met de echte levenscyclus van een service. Deze aanpak maakt het ook moeilijk om delen van een levenscyclus binnen één service te hergebruiken voor een andere applicatie. Hoofdstuk 3 stelt het "reactive pattern" voor om dit probleem op te lossen. Dit laat systeembeheerders toe voor iedere applicatie een levenscyclus op maat te maken. Deze aanpak maakt het ook mogelijk om delen van een levenscyclus te groeperen in een herbruikbare bibliotheek die gedeeld kan worden met andere systeembeheerders. Dit hoofdstuk eindigt met een evaluatie van het ecosysteem dat ontstaan is doorheen de twee jaar na initiële uitgave van het "charms.reactive" framework, dat het reactive pattern implementeert. Deze evaluatie toont aan dat deze aanpak inderdaad samenwerking tussen systeembeheerders en het delen van code stimuleert.

De derde moeilijkheid draait rond het vastleggen en hergebruiken van een kennis die een systeembeheerder heeft over hoe microservice applicaties opgebouwd worden. Cloud-native beheersprogrammas zoals Kubernetes bieden geen effectieve manier aan om de communicatie tussen microservices expliciet te modelleren en te configureren. Hoofdstuk 4 stelt "orcon" voor om dit probleem aan te pakken. Dit is een extensie van Kubernetes die systeembeheerders de mogelijkheid bied om relaties tussen microservices te modelleren. De extensie gaat dan de communicatie tussen die microservices configureren en automatisch aanpassen indien er wijzigingen zijn. De evaluatie in dit hoofdstuk toont aan dat de extensie relaties tussen microservices automatisch kan opzetten en updaten zonder dat er interactie tussen mensen nodig is. De evaluatie toont ook dat deze herconfiguratie gebeurt in maar 44 milliseconden per service, wat veel beter is dan vergelijkbare beheerstools.

De vierde moeilijkheid zit in het beheer van dataverwerking stromen die over organisatorische grenzen heen gaan. Serverless platformen hebben veel potentieel om het beheer van dataverwerking stromen te vereenvoudigen. Deze platformen bieden echter geen effectieve tools aan om met meerdere individuele teams éénzelfde datastroom te beheren. Hoofdstuk 5 stelt "Plumber" voor om dit probleem op te lossen. Verschillende onafhankelijke partijen kunnen door middel van Plumber samen een dataverwerking stroom ontwikkelen en updaten. Het biedt een gebruiksvriendelijke interface aan en ondersteunt geavanceerde eigenschappen zoals atomaire updates van een draaiende opstelling zonder dataverlies. Deze maakt het ook mogelijk om vorige versies van de verwerkingscode eenvoudig terug te zetten. De evaluatie op het einde van dit hoofdstuk toont dat een draaiende verwerkingsstroom kan geüpdatet worden in 12 seconden zonder enig dataverlies.

De vijfde en laatste moeilijkheid die in deze thesis behandeld wordt, gaat over het beheren van applicaties die op een mix van cloud en edge infrastructuur draaien. Deze mix, die de "fog" genoemd

wordt, zorgt er voor dat een aantal assumpties van bestaande beheerstools en methodologieën niet meer kloppen. Er is dus een nood aan het aanpassen van deze tools en methodologieën om met deze nieuwe omgeving te kunnen omgaan. Om dit probleem aan te pakken introduceert Hoofdstuk 6 een visie van een "fog native" methodologie. Het idee van intentie-gebaseerd beheer van applicaties staat hier centraal. Dit is een radicale verandering tegenover het traditionele idee van declaratief beheer van services. In plaats van een declaratieve beschrijving van de gewenste staat van de applicatie, werkt dit met een beschrijving van de intentie van de applicatie. Dit geeft het systeem de mogelijkheid om zelf de gewenste staat aan te passen op basis van welke infrastructuur op dat moment beschikbaar is en waar de gebruikers van een applicatie zitten. Dit hoofdstuk eindigt met een evaluatie die aantoont wat de impact is van microservice applicaties te distribueren over een mix van cloud en edge infrastructuur.

De methodes en platformen voorgesteld in deze thesis pakken een aantal belangrijke problemen aan op het gebied van beheer van applicaties. Daarnaast geeft dit onderzoek ook een leidraad om de uitdagingen rond het beheer van services in de fog aan te pakken. Toekomstig onderzoek kan, bijvoorbeeld, de aanpak van de orchestrator conversation gebruiken om services in de fog eenvoudiger te beheren. De overhead die gepaard gaat met dergelijke oplossingen kan aangepakt worden door on-demand besturingssoftware te ontwikkel door middel van WebAssembly. Een tweede interessante toekomstige onderzoekspiste is om samenwerking bij Kubernetes controllers aan te sporen, door het reactive pattern te implementeren voor Kubernetes. Er zijn ook oplossingen nodig om de vertrouwelijkheid en veiligheid van services in de fog de verzekeren, bijvoorbeeld door middel van "device attestation" en "confidential computing". Als laatste zijn er een aantal open onderzoeksvragen betreffende intentie-gebaseerd beheer van services, zoals het vertalen van intentie-modellen naar gewenste-staat modellen, en het gebruik van kunstmatige intelligentie voor verdere optimalisaties.

Summary

Collaboration across organizational boundaries, whether it is between teams, organizations or companies, is becoming increasingly important. The fourth industrial revolution is one of the driving factors for this because it is characterized by widespread interconnection of cyber-physical systems. As an example, an industry 4.0 manufacturing company might interconnect each manufacturing machine on a production line so they can share data, learn, and adapt their behavior. Moreover, this production line would be connected to other parts of the organization to automatically report and integrate into various IT systems. Furthermore, the entire IT system of the company would connect to IT systems of other companies in their supply chain in order to ensure further integration.

Complexity of IT systems increases as digital transformation touches more and more parts of society. Applications become composed systems consisting of multiple components interacting with each other to achieve the required objective. Although microservice architecture is often positioned as a way to address the complexity of monolithic applications, it also increases complexity by introducing additional communication between microservices and additional frameworks to manage them. The rising complexity of composed applications makes them increasingly challenging to deploy and operate. As such, there are a number of open questions in the field of service orchestration, which investigates the process of designing, deploying and managing such applications.

The first difficulty is in capturing and reusing a system administrator's knowledge about when to deploy what. Current state of the art configuration management tools are not great at encapsulating knowledge because they are based on declarative models. Although these models are great at describing the desired state of an application, they cannot be used to create logic that decides what that desired state should look like. Solving this challenge means system administrators can more easily share knowledge and create software that helps them better manage the complexity of composed applications. Chapter 2 proposes the orchestrator conversation as a way for system administrators to capture their knowledge about when to deploy what. It introduces the concept of an "orchestration agent" as a way to translate abstract higher-level models into more concrete lower-level models. Similarly, while the application is running, orchestration agents also translate low-level information about the running infrastructure and services into a higher-level abstract status of the complete application. Evaluation of a prototype implementation shows this approach is able to translate higher-level abstract models into lower-level concrete models with minimal overhead.

The second challenge is capturing and reusing a system administrator's knowledge about when

to perform which management actions during the lifecycle of an application. Declarative models can only capture part of a system administrator's tasks: after the application reaches the desired state, they still require frequent maintenance and updates in order to ensure their correct execution. Therefor, cloud modeling languages like the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) allow a system administrator to define management actions in a declarative workflow model. The orchestrator then decides when to run which actions based on an internal predefined lifecycle. This approach is very limited, however, since all applications are locked into the lifecycle supported by the orchestrator. Moreover, this approach makes it difficult to reuse actions across multiple services. Chapter 3 proposed the reactive pattern to make it easier to create custom lifecycles to manage a single component of a composed application. Moreover, it is specifically designed to enable an ecosystem where developers share reusable collections of lifecycle steps to use between services. The reactive pattern is an event-based system where each management action defines their pre- and post-conditions using "flags". Using these properties, actions are chained into emergent workflows which manage the complete lifecycle of a service. The chapter ends with a reflection on more than two years of using this pattern in production as part of the charms.reactive framework built for the Juju orchestrator. This shows the pattern indeed fostered an ecosystem of sharing.

The third difficulty concerns encapsulating and reusing system administrator's knowledge about composing cloud-native microservice applications. Although cloud-native orchestrators such as Kubernetes provide battle-tested APIs to manage containerized applications, they fail to provide comprehensive methods to automatically manage dependencies between individual microservices. As a result, such dependencies are often managed using ad-hoc solutions that provide very little design-time insight and are prone to break. Chapter 4 addresses this challenge by proposing orcon, an extension to the Kubernetes API inspired by the orchestrator conversation. It allows declarative modelling and automatic (re-)configuration of dependencies between microservices. It does this in a Kubernetes-native way so it integrates with the existing ecosystem of tools for Kubernetes. A functional evaluation shows orcon removes the need for human-to-human interactions when microservice dependencies change. A performance evaluation shows it takes 0.44 seconds per service to propagate changes in dependencies, greatly outperforming traditional agent-based orchestrators using cloud modeling languages.

The fourth difficulty concerns the management of stream processing pipelines that cross organizational boundaries. Although the serverless paradigm aid in this endeavor, they lack in support for cross-domain collaboration. Specifically, they do not provide cohesive facilities for multiple collaboratively develop a stream processing pipeline where each party is responsible for a different part of the pipeline. Chapter 5 addresses this challenge by proposing Plumber, a framework for building and running serverless stream processing pipelines that cross organizational borders. It has a specific focus on enabling collaborative creation of such pipelines, a focus that is lacking in current state of the art. It goes beyond that focus, however, to deliver a user-friendly UI and advanced features such as atomic upgrades, automatic scaling and seamless roll-back to previous versions. Like the orcon solution from Chapter 4, it is also a Kubernetes-native framework ensuring high compatibility with the existing ecosystem. The evaluation at the end of this chapter shows it can update a running cross-domain analytics pipeline in 12 seconds without any data loss.

Finally, the fifth difficulty concerns managing applications that run on a mixture of cloud and edge

resources called “fog computing”. This breaks a number of assumptions of traditional and cloud native service orchestration approaches. To solve this issue, the practices to deploy and manage services must adapt to this new reality of a cloud-edge continuum. Chapter 6 proposes a vision for addressing this challenge by introducing a “fog native” architecture and a set of design patterns. The architecture is centered around *intent-based workflow construction*. This is a radical change compared to the traditional desired-state based methods for managing applications. Instead, developers specify the desired *behavior* of the application using *intents*. A workflow manager interprets these intents and creates a desired state model which ensures the desired behavior. This gives the system flexibility to dynamically update the desired state based on user demand, location and available infrastructure. It also introduces the concept of a Fog mesh, which handles communication between microservices and to external users. The chapter ends with an evaluation showing the impact of running microservice-based applications in a fog ecosystem.

To conclude, the approaches proposed in this dissertation address important challenges in the field of service orchestration in the cloud, and carve a path towards facilitating service orchestration in the fog. Future research can build on this work by, for example, adapting the agent-based orchestration approach of Chapter 2 to the fog. Addressing the overhead of agent-based approaches might be possible by creating on-demand control planes using technologies such as WebAssembly. A second direction for future research is to improve collaboration in the creation of Kubernetes controllers, for example by adapting the reactive pattern to Kubernetes. Furthermore, the realities of the cloud-edge continuum demand new approaches to security of infrastructure and workloads. Novel approaches are needed to ensure service orchestration platforms integrate with device attestation methodologies and confidential computing. Finally, there are a number of open challenges concerning intent-based approaches for service orchestration. Research is needed into the structure and form of intent models, translating intent models to desired-state models, and using AIOps to further optimize service orchestration in the fog.

1 Introduction

"The operating mechanism [of the analytical engine] can even be thrown into action independently of any object to operate upon (although of course no result could then be developed). Again, it might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."

- Ada Lovelace, 1843

1.1 Collaborative Compositions

Collaboration across organizational boundaries, whether it is between teams, organizations or companies, is becoming increasingly important. Gartner predicts that by 2023, organizations which have the technology and processes to enable inter-enterprise data sharing will outperform those that do not [8]. One of the driving factors for this is the fourth industrial revolution, characterized by widespread automation and interconnection of cyber-physical systems. Not only are the technological challenges rooted in enabling collaboration [6], but collaboration is also a key factor in adopting industry 4.0 across sectors [17]. As an example, in an industry 4.0 factory,

manufacturing machines on a production line are cyber-physical systems that communicate detailed properties of each produced component to each other. This allows each machine to slightly adapt its behavior to reduce production faults or to create slight variations of the same product. Furthermore, by integrating the IT systems of multiple companies across the value chain of a product, machines can take into account even more details of a component, and entire production lines can change their behavior based on the behavior and requirements of other entities in the chain. As such, the value of the fourth industrial revolution becomes even more apparent when multiple companies across a value chain collaborate to set this transition in motion [17].

Complexity of IT systems increases as digital transformation touches more and more parts of society. Applications become composed systems consisting of multiple components interacting with each other to achieve the required objective. For example, in 2018, a survey of CIOs reported a single web or mobile transaction would cross an average of 35 different technology systems or components [9]. This complexity negatively impacts service reliability, performance [9], and release time. Although microservice architecture is often positioned as a way to address the complexity of monolithic applications, it also increases complexity by introducing additional communication between microservices and additional frameworks to manage them [11].

1.2 Facilitating Service Orchestration

The rising complexity of composed applications makes them increasingly challenging to deploy and operate. This management of applications is referred to as *service orchestration* in the context of this work, in line with the taxonomy of Weerasiri et al. [25]. It encompasses the entire process of selecting, describing, deploying, configuring, monitoring and controlling the infrastructure and services that make up that application.

Configuration management is one of the first efforts to facilitate this process. Burgess et al. introduced the concept of convergence towards a desired state, also called the "desired state princple": a system administrator defines the desired state of the application in a declarative model, and the system iteratively changes the application to get to this desired state [5]. One of the advantages of this method is that it is more resilient because the system makes no assumptions about the initial state of the application. This is in contrast to managing applications using imperative shell scripts which inherently make a lot of assumptions about the initial state of the system before they run. A second advantage of this approach is that the current state of an application is easily visible and auditable using the declarative model.

Infrastructure as Code is the next step of this paradigm, where both the application and the underlying infrastructure are described using *cloud modeling languages* [4]. Many of these languages embrace the composed nature of modern applications by describing them as topologies of resources, services and relationships. Although most tools, such as Juju [7], use a custom modeling language [4], there are also efforts to create a standardized modeling language such as the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [18].

The advent of containerization creates an interesting shift in service orchestration. By containing each service inside of a container, the scope of service orchestration is reduced to *container orchestration*. Individual services do not have to be installed and configured anymore. Instead, developers provide standardized containers in which these services are already preinstalled. This effectively disconnects the challenge of installing and configuring a single service from that of orchestrating multiple services. As a result, container orchestrators such as Kubernetes can treat every single container in a standardized manner, regardless of what service and technologies are running inside of it.

1.3 From Cloud to Edge

The question of where these applications are running is changing rapidly. The industry as a whole has been going through an extensive transition from private data centers and on-premise servers to the cloud. As such, the nature of software development has changed to adapt to this trend, creating new paradigms such as *cloud native* and *serverless*. The *cloud native* paradigm, centers around turning monoliths into microservices [12] designed from the ground up to take advantage of the benefits of the cloud [2, 16]. This allows teams to release faster, increase reliability, and expedite operations by taking full advantage of cloud resources and their elasticity. Serverless is a paradigm that simplifies operating an application by providing an opinionated abstraction over all layers except the core business logic [3]. Users only need to supply their business logic and minimal configuration to create a running, production-ready application. Many operational concerns, such as scaling, scheduling, and observability, are taken care of by the serverless platform [10]. Simultaneously, serverless approaches have the potential to reduce resource usage [1][20]. For this reason, serverless is also gaining traction in the IoT settings [24][21], and in the edge with offerings such as Cloudflare Workers [23].

At the same time, however, more companies are combining cloud applications with edge computing [15]. The Netflix Open Connect program, for example, invites ISPs to place Netflix caching servers in the edge, in order to improve user experience and decrease strain on the network. The cloud and edge have historically been regarded as vastly different ecosystems, with vastly different approaches to developing, deploying and managing applications. The idea of *"fog computing"* or the "cloud-edge continuum" shown in Figure 1.1 aims to break this dogma by extending the concepts and methodologies of the cloud towards the edge. The resulting fog allows developers to seamlessly run applications on a mixture of cloud and edge resources. This enables them to increase privacy [26] and reduce latency [19], and helps ISPs to ensure more efficient resource usage.

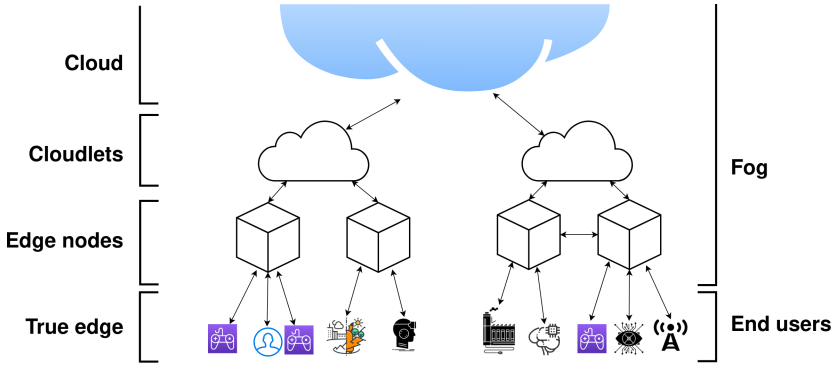

Figure 1.1: The fog extends from the cloud to the edge, creating a single cloud-edge continuum on which applications can be deployed.

1.4 Research Questions

1.4.1 Deciding what should be deployed

Encapsulating and reusing knowledge is difficult using the state of the art in configuration management tools. This issue stems from the foundational theory behind this field: the idea that infrastructure code should be a declarative description of the desired end-state of your application, as posited by Burgess et al. [5]. Declarative models are great at describing the desired end result. Where they fall short, however, is in describing how to decide what the end result should be. As a result, declarative models are great at encapsulating a system administrator's knowledge about how to deploy something, but are insufficient for encapsulating the knowledge about deciding what to deploy. As such, the first research question investigates how to address this limitation. Solving this challenge means system administrators can more easily share knowledge and create software that helps them better manage the complexity of composed applications.

RQ 1. *How to encapsulate and reuse system administrator's knowledge about when to deploy what?*

1.4.2 Deciding when to run management actions

Declarative models allow a system administrator to codify the desired end state of a service. After initial deployment, however, there are a number of *day-2 operations* which need to be performed such as maintenance, integration and updates. Therefor, cloud modeling languages like TOSCA allow a system administrator to specify management and orchestration actions in a declarative workflow model. The orchestrator defines a lifecycle for each service, and the workflow model specifies for each step in the lifecycle what action should be taken. This workflow model allows a system administrator to encapsulate their knowledge about how to manage a service. This approach has two issues, however. The first being that the lifecycle provided by the orchestrator

does not match the actual lifecycle of the managed service. In reality, many services have much more lifecycle steps than those provided by the orchestrator. The second issue resides in the reuse of individual aspects of a service's lifecycle. Many services share multiple lifecycle steps, but the knowledge about when to perform that step cannot be reused across multiple services. As a result, creating a lifecycle generally requires complete knowledge of when each individual step of each aspect of the service should be taken.

RQ 2. *How to encapsulate and reuse system administrator's knowledge about when to perform which management actions?*

1.4.3 Deciding how to connect microservices

Microservice architecture fully commits to the idea of an application as a composition of small, independent services. This revolution resulted in a number of interesting solutions for orchestrating such an application, with Kubernetes being the most widely used one. Although Kubernetes provides very flexible APIs to manage containerized applications, it fails at providing comprehensive explicit management of dependencies between individual microservices of an application. As a result, such dependencies are often managed using ad-hoc solutions that provide very little design-time insight and are prone to break. Although service meshes aim to solve a number of operational issues concerning dependencies between services, they do not allow explicit modelling and automatic configuration of these dependencies. This makes it challenging for system administrators to encapsulate their knowledge about how to configure dependencies between microservices and how to compose microservices to get the desired application.

Although some cloud modeling languages support encapsulating this knowledge about regular services, adapting them to microservice applications results in sub-optimal solutions. The first issue is that these adaptations do not integrate into the wider cloud native ecosystem; they just create a new abstraction layer on top of existing orchestrators, greatly limiting compatibility with existing tools. Secondly, since these tools have not been developed for containerized microservice scenarios, they often have an unacceptable per-microservice overhead which is sometimes larger than the actual running microservice itself.

RQ 3. *How to encapsulate and reuse system administrator's knowledge about composing microservice applications and (re-)configuring their internal dependencies, in a way that fully integrates into a cloud native ecosystem?*

1.4.4 Deciding how to process cross-domain streams

Serverless platforms aim to make it easier for developers to develop and deploy cloud services. Applied to stream processing, users only need to supply their business logic and minimal configuration to form a running topology of stream processing code. One area where these platforms still lack, however, is in supporting cross-domain collaboration. The current state of the art in stream processing platforms do not offer cohesive facilities for multiple independent parties to

collaboratively develop a stream processing pipeline where each party is responsible for a different part of the pipeline. Existing technologies lack methods for collaboratively modelling and building data processing topologies and lack hands-off change management techniques.

RQ 4. *How can multiple independent parties collaboratively create serverless streaming pipelines?*

1.4.5 Bringing cloud native to the fog

With the advent of fog computing, microservices and serverless functions are now not necessarily only running in the cloud anymore. Instead, computation is spread across the cloud-edge continuum, running everywhere from the cloud to the edge. This means the practices to deploy and manage services must adapt to this new reality. While the cloud native paradigm's objectives of faster releases, increased reliability and expedited operations are very useful in the fog, the paradigm itself is grounded in a number of assumptions which do not hold true anymore in that environment.

RQ 5. *How to adapt the cloud native paradigm to the cloud-edge continuum of the fog?*

1.5 Research Contributions and Outline

The research that is part of this dissertation aims to facilitate the management of collaborative, composed applications in the cloud and the edge. The main contributions made in this area are the following.

1.5.1 Orchestrator Conversation (Chapter 2)

Chapter 2 proposes the "orchestrator conversation", a hierarchical agent-based solution for turning high-level requests for desired applications into low-level declarative models describing the required services to meet that request. This is in line with RQ 1. The orchestrator conversation addresses the challenges of creating new abstractions using cloud modeling languages by introducing the concept of an "orchestration agent". Each orchestration agent serves as an encapsulation of knowledge about how to translate high-level abstract request models into lower-level more concrete desired state models. Similarly, while the application is running, orchestration agents also translate low-level information about the running infrastructure and services into a higher-level abstract status of the complete application.

The chapter defines a number of concepts pertaining to how the orchestrator conversation interacts internally and externally:

- Internal communication happens using declarative "request models" to define the desired state and "runtime models" to describe the current state of the infrastructure and application.

- "Service agents" are used by the orchestrator conversation to communicate with lower level orchestrators and configuration management systems.
- "Collaborator relationships" are used by service agents to collaborate in a non-hierarchical fashion.
- "Operator relationships" are used by orchestrator agents to drive the behavior of service agents in a hierarchical fashion.

Finally, the chapter presents a prototype implementation of the orchestrator conversation and a functional and performance evaluation. The functional evaluation shows the orchestrator conversation succeeds in allowing a user to deploy a Hadoop cluster using a much simpler and abstract model compared to the state of the art. The performance evaluation shows the overhead of translating the higher-level abstract model into lower-level concrete models (100 milliseconds) is negligible compared to the time to deploy the application using the Juju orchestrator (20 minutes).

1.5.2 The reactive pattern (Chapter 3)

Chapter 3 presents the "reactive pattern", a method to create emergent workflows to manage individual services. This is in line with RQ 2: the reactive pattern addresses the shortcomings of managing services using lifecycles by creating a flexible event-based system where lifecycles emerge from the pre- and post-conditions of individual management actions. Each management action defines a number of "flags" which need to be on or off, in order for the management action to run. While an action is running, it turns flags on or off, which might trigger other actions to run afterwards. Since actions have no direct dependencies on each other, only on flags, they can be grouped into "layers" regardless of the structure of the emergent workflow. As such, it becomes possible to group handlers by which aspect of the service they manage. This makes it possible to create an ecosystem of reusable layers, each of which contains management actions for a different aspect of a service.

The chapter also discusses the "charms.reactive" framework, which implements this pattern on top of Juju's declarative workflows, which are encapsulated in "Charms". It evaluates the pattern by investigating the ecosystem of Charms and layers that has arisen during two years after the initial release of the framework. The results show 1/3 of all active Charms use this framework, 67% of those share parts of their management workflow with at least one other Charm and 73% of those share parts of their relationship workflow with at least one other Charm.

1.5.3 Orcon (Chapter 4)

Chapter 4 presents "orcon", an extension to Kubernetes allowing users to model and manage dependencies between microservices, addressing RQ 3. The chapter introduces a definition for what a service relationship is, and shows a proof-of-concept Kubernetes controller that adds this functionality to Kubernetes. The resulting extension is completely Kubernetes-native: it adds this

functionality to the existing Kubernetes API so that it is compatible with the existing ecosystem of tools which use this API.

The chapter also presents a functional and performance evaluation of this prototype. The functional evaluation shows the automatic change management of modeled dependencies removes the need for human-to-human interaction when changes on one side of the dependency need configuration adaptations on the other side. The functional evaluation also shows the existing Kubernetes API is not abstracted to users, and thus all Kubernetes functionality and compatibility is available. The performance evaluation shows change propagation happens with an overhead of only 0.44 seconds per service, greatly outperforming traditional agent-based orchestrators using cloud modeling languages.

1.5.4 Plumber (Chapter 5)

Chapter 5 presents "Plumber", a serverless framework that supports creating cross-organizational stream processing pipelines, answering RQ 4. The framework also offers robust change management of streaming topologies with the following features.

- Declarative definition of pipelines and seamless rollback to previous versions.
- Horizontal autoscaling out of the box.
- Atomic no-touch upgrades from the standpoint of the stream such that there is a single upgrade point after which all new messages are processed by the upgraded topology.
- At-least-once processing semantics at all times.
- Kubernetes-native: all interaction with Kubernetes happens using desired state models of the Kubernetes API.

The chapter also presents a functional and performance evaluation of this framework. The functional evaluation shows the platform makes it possible to collaboratively create stream processing pipelines that cross organizational borders. The performance evaluation shows the framework can update a running cross-domain analytics pipeline in 12 seconds without any data loss or duplication.

1.5.5 Fog-native architecture (Chapter 6)

Chapter 6 introduces a vision for a "fog native" paradigm and architecture which helps developers create applications that run on a mixture of cloud and edge resources. This addresses RQ 5. The chapter identifies four fundamental assumptions of the cloud native paradigm which do not hold true in the fog, and proposes an architecture and a set of design patterns to remedy these issues. Key aspects of the fog native paradigm the following.

The paradigm is centered around *intent-based workflow construction*. This is a radical change compared to the traditional desired-state based methods for managing applications. Instead,

developers specify the desired *behavior* of the application using *intents*. A workflow manager interprets these intents and creates a desired state model which ensures the desired behavior. This gives the system flexibility to dynamically update the desired state based on user demand, location and available infrastructure.

The chapter also proposes the concept of a *fog mesh*, providing three distinct functionalities.

- Similar to API gateways in a cloud native environment, the fog mesh handles **external connectivity** from users to microservices. Each fog mesh proxy acts as an ingest point, which makes it possible to distribute API gateway functionality in the edge, closer to users.
- Apart from the data plane, the fog mesh proxies are also control-plane ingest points for the system in that they receive end user requests for workflows and communicate them to the workflow manager. This has the advantage of aggregating workflow requests to avoid overloading the workflow manager each time individual end users change.
- The fog mesh also takes up the role of traditional service meshes, meaning they facilitate **inter-microservice connectivity**. In contrast to traditional service meshes, however, the fog mesh does not require a sidecar proxy for each single microservice. Instead, microservices are grouped in regional clusters based on the network topology and demand. Each regional cluster has a single service mesh proxy handling inter-microservice communication for the entire cluster.

Finally, the chapter provides an evaluation showing the impact of running microservice-based applications in a fog ecosystem, confirming, for example, network latency plays a bigger part in distributed workflow response time in the fog compared to the cloud.

1.5.6 Conclusion (Chapter 7)

Finally, Chapter 7 concludes this thesis, reflects on lessons learned during the course of this PhD, and looks forward to propose interesting future research directions.

1.6 Research Projects

Parts of the research conducted during this PhD are the result of different national and international research projects which are listed below:

1. **Fed4FIRE (FP7-ICT):** *"Federation for FIRE"*. This project established a European Federation of research testbeds by connecting over 23 instances all across Europe. This allows a researcher to provision and connect physical and virtual infrastructure such as servers, virtual machines, wireless and wired networks. As part of this project, IDLab created the first iteration of the Tengu testbed, which allowed easy provisioning of data processing platforms such as Apache Hadoop and Apache Spark. This testbed has been used extensively by national and international research projects.

2. **MmmooOgle (bilateral agreement Bovicom - iMinds):** This Smart Farming project investigated how Big Data technology could be used to combine and process data streams of varying sensors in a modern dairy farm. As part of this project, iMinds helped Bovicom explore the Big Data landscape and set up a proof-of-concept data analytics stream which predicted the fertility of cows.

3. **DeCoMAdS (VLAIO SBO):** *Deployment and Configuration Middleware for Adaptive Software-as-a-Service:* The goal of this project was to support Software-as-a-Service providers in creating adaptive, scalable and performant cloud applications. One of the goals was to create a middleware solution to aid SaaS vendors in creating their platforms. This project had a large advisory board of industry partners to bring use-cases and orient the research.

4. **Providence+ (iminds.icon):** The goal of this project was to optimize news publication strategies by acting on predictions of the virality of news stories. As part of this project IDLab created a platform to ingest raw click and scroll data from the news websites MediaMonkey and VRT, and to predict which stories would "become viral".

5. **AMiCA (VLAIO SBO):** This project had as goal to identify possibly threatening situations on social networks by means of text and image analysis. This is to ensure the online safety of children. As part of this project, IDLab created a platform which ingests a real time stream of tweets from Twitter to analyze and highlight cases of cyberbullying, sexually transgressive behavior, depression and suicidal behavior.

6. **City of Things (imec):** This is an umbrella program for a number of projects that investigate how hardware and software can be combined to create smart-city solutions. The city of Antwerp, Belgium, is home to a digital testing ground for these projects. Parts of this research were conducted as part of a number of sub-projects, such as observing air quality using sensors mounted on delivery vehicles, and supporting the development of scalable applications on IoT data using the Obelisk IoT platform.

7. **Digital Cinema (VLAIO O&O):** The goal of this project was to create a distributed platform for managing movies in a cinema. Traditionally, movies are stored on centralized hardware and downloaded by the cinema projector in each room when needed. As part of this project, IDLab created a swarm-based platform which allows the cinema projectors to collectively store movies without a centralized platform. Using a combination of gossip protocols and promise theory, the projectors collectively decide how and where to store which movies.

8. **Solid Web Monetization (Grant for the Web):** This project created a method for Solid web applications to monetize content without ads. Solid is a new web standard for building privacy-preserving and interoperable web applications. Since Solid web applications cannot depend on data collection as their business model, this project investigated how to incorporate Web Monetization into the Solid ecosystem. With Web Monetization, creators can request micropayments for viewing their content in a browser. IDLab created a prototype framework for monetizing content in a privacy-preserving way and drafted a new

W3C standard which enables this functionality in an interoperable way.

9. **Fed4FIRE+ (Horizon 2020):** This project continues the effort of the original Fed4FIRE in creating and connecting research testbeds across Europe. As part of this project, IDLab created "Tengulabs", the second iteration of the Tengu testbed which allows researchers to easily provision Kubernetes clusters and deploy applications to them.
10. **iCosy2 (VLAIO O&O):** This project aims to improve indoor comfort by combined control of ventilation and sun protection systems from Renson. As part of this project, IDLab designed architectures for cloud-based and edge-based machine learning systems for combined control. IDLab also created a prototype cloud-based architecture to evaluate the cost and feasibility of such a solution.

1.7 Publications

The results of the research during this PhD have been published in scientific journals and presented at different international conferences. This section provides an overview of these publications.

1.7.1 Publications in international journals

1. T. Vanhove, **M. Sebrechts**, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, **Data transformation as a means towards dynamic data storage and polyglot persistence**, *International Journal of Network Management*, vol. 27, no. 4, p. e1976, 2017, doi: 10.1002/nem.1976.
2. **M. Sebrechts**, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, **Orchestrator conversation: Distributed management of cloud applications**, *International Journal of Network Management*, vol. 28, no. 6, p. e2036, 2018, doi: 10.1002/nem.2036.
3. J. Santos, T. Vanhove, **M. Sebrechts**, T. Dupont, W. Kerckhove, B. Braem, G. Van Seghbroeck, T. Wauters, P. Leroux, S. Latre, B. Volckaert, F. De Turck, **City of Things: Enabling Resource Provisioning in Smart Cities**, *IEEE Communications Magazine*, vol. 56, no. 7, pp. 177-183, Jul. 2018, doi: 10.1109/MCOM.2018.1701322.
4. V. Bracke, **M. Sebrechts**, B. Moons, J. Hoebeke, F. De Turck, and B. Volckaert, **Design and evaluation of a scalable Internet of Things backend for smart ports**, *Software: Practice and Experience*, vol. 51, no. 7, pp. 1557-1579, 2021, doi: 10.1002/spe.2973.
5. **M. Sebrechts**, S. Borny, T. Wauters, B. Volckaert, and F. De Turck, **Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes**, *IEEE Access*, vol. 9, pp. 133387-133401, 2021, doi: 10.1109/ACCESS.2021.3115438.
6. **M. Sebrechts**, B. Volckaert, F. De Turck, K. Yangy, and M. AL-Naday, **Fog Native Architecture: Intent-Based Workflows to Take Cloud Native Towards the Edge**, *IEEE Communications Magazine*, pp. 1-7, Aug. 2022, doi: 10.1109/MCOM.003.2101075.

1.7.2 Publications in international conferences

1. **M. Sebrechts**, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, **Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration**, *in 2016 IEEE International Conference on Mobile Services (MS)*, Jun. 2016, pp. 156-159. doi: 10.1109/MobServ.2016.31.

2. **M. Sebrechts**, S. Borny, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, F. De Turck, **Model-driven deployment and management of workflows on analytics frameworks**, *in 2016 IEEE International Conference on Big Data (Big Data)*, Dec. 2016, pp. 2819-2826. doi: 10.1109/BigData.2016.7840930.

3. **M. Sebrechts**, G. Van Seghbroeck, and F. De Turck, **Optimizing the Integration of Agent-Based Cloud Orchestrators and Higher-Level Workloads**, *in Security of Networks and Services in an All-Connected World*, 2017, pp. 165-170, doi: 10.1007/978-3-319-60774-0_16.

4. **M. Sebrechts**, C. Johns, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, **Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications**, *in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 326-333. doi: 10.1109/CLOUD.2018.00048.

5. T. Goethals, **M. Sebrechts**, A. Atrey, B. Volckaert, and F. De Turck, **Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications**, *in 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, Nov. 2018, pp. 1-8. doi: 10.1109/SC2.2018.00008.

6. T. Goethals, S. Kerkhove, L. V. Hoye, **M. Sebrechts**, F. Turck, and B. Volckaert, **FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers**, *in the 9th International Conference on Cloud Computing and Services Science (CLOSER)*, 2019. doi: 10.5220/0007706000900099.

7. **M. Sebrechts**, T. Goethals, T. Dupont, W. Kerckhove, R. Taelman, F. De Turck, B. Volckaert, **Solid Web Monetization**, *in the 22st International Conference on Web Engineering (ICWE 2022)*, 2022, doi: 10.1007/978-3-031-09917-5_40.

8. T. Goethals, **M. Sebrechts**, M. Al-Naday, B. Volckaert, and F. De Turck, **A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing**, *in 2022 IEEE International conference on Edge Computing (EDGE)*, Jul. 2022.

9. **M. Sebrechts**, T. Ramlot, S. Borny, T. Goethals, B. Volckaert, and F. De Turck, **Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI**, *Submitted for review*, 2022.

1.8 Code Repositories

Applications and code developed as part of this PhD have been published using open source licenses. This includes applications, frameworks, evaluation code and simulation code. The following list provides an overview of all public code repositories created as part of this PhD.

Note that this list does not include all code contributions of this PhD. Changes to existing open source projects that have been submitted and accepted to their respective repositories are not listed here.

1. The old Tengu testbed: a Juju-based platform to deploy and manage data analytics platforms. This organization contains a number of repositories with open source components of this platform.
   `https://github.com/IBCNServices/tengu-docs`.

2. Public documentation of the old Tengu testbed.
   `https://github.com/IBCNServices/tengu-docs`.

3. A Docker container based on the charmbox that can be used as Eclipse Che workspace to develop Juju Charms in the Eclipse Che IDE.
   `https://github.com/IBCNServices/che-charmbox`.

4. Orchestrator Conversation simulation code and results.
   `https://github.com/IBCNServices/oa`.

5. Evaluation code and evaluation results of the charms.reactive framework.
   `https://github.com/IBCNServices/reactive-pattern-results`.

6. orcon: a Kubernetes service relationship orchestrator.
   `https://github.com/IBCNServices/`.

7. orcon evaluation code and benchmark results.
   `https://github.com/IBCNServices/kubernetes-relationships-results`.

8. Easy OpenVPN Server: snap package of OpenVPN server with automated certificate management.
   `https://github.com/IBCNServices/easy-openvpn-server`.

9. Public documentation of the Tengulabs Fed4FIRE+ testbed.
   `https://github.com/IBCNServices/Tengulabs`.

10. Plumber: platform for collaboratively managing serverless streaming applications on Kubernetes. Contains the application code, evaluation code and benchmark results.
    `https://github.com/IBCNServices/plumber`.

1.9 Awards and Recognition

Parts of the research conducted during this PhD have received awards and recognition:

1. **Master's thesis recognition (2015)**: The master thesis titled "Development of a scalable and modular PaaS for Big Data solutions" was the start of this research. It received recognition as a finalist in the ie-net awards for best Flemish engineering master theses. The thesis was also longlisted for the Agoria award for best STEM thesis.
2. **Best Student Paper Award SC2 2018**: The paper "Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications" [14] received this award at the 8th IEEE International Symposium on Cloud and Services Computing.
3. **Best Student Paper Award CLOSER 2019**: The paper "FUSE: a Microservice approach to Cross-domain Federation using Docker Containers" [13] received this award at the 9th International Conference on Cloud Computing and Services Science.
4. **Ubuntu Member (2021):** Due in part to his contribution to the Juju and Snapcraft ecosystems as part of this research, Merlijn received the "Ubuntu Member" title, for "significant and sustained contribution to Ubuntu". Juju is an open source cloud modeling language and orchestrator, and Snapcraft is an open source package manager created by Canonical, the company behind Ubuntu.
5. **Best Demo Paper Award ICWE 2022:** The demo paper "Solid Web Monetization" [22] received this award at the 22nd International Conference on Web Engineering.

Bibliography

[1] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 884–889, New York, NY, USA, August 2017. Association for Computing Machinery. doi:10.1145/3106237.3117767.

[2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In Antonio Celesti and Philipp Leitner, editors, *Advances in Service-Oriented and Cloud Computing*, Communications in Computer and Information Science, pages 201–215, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-33313-7_15.

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless Computing: Current Trends and Open Problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, Singapore, 2017. URL: https://doi.org/10.1007/978-981-10-5026-8_1.

[4] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys*, 51(1):22:1–22:38, February 2018. doi:10.1145/3150227.

[5] Mark Burgess and Oslo College. Cfengine: a site configuration engine. In *USENIX Computing systems, Vol*, 1995.

[6] Luis M. Camarinha-Matos, Rosanna Fornasiero, and Hamideh Afsarmanesh. Collaborative Networks as a Core Enabler of Industry 4.0. In Luis M. Camarinha-Matos, Hamideh Afsarmanesh, and Rosanna Fornasiero, editors, *Collaboration in a Data-Rich World*, IFIP Advances in Information and Communication Technology, pages 3–17, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-65151-4_1.

[7] Canonical Ltd. Ubuntu Juju: Operate big software at scale on any cloud, 2017. URL: https://jujucharms.com/.

[8] Alan D. Duncan. Over 100 Data and Analytics Predictions Through 2025. Technical Report G00744238, Gartner, Inc., March 2021.

[9] Dynatrace LLC. Lost in the Cloud? Top challenges CIOs face amidst their complex enterprise cloud ecosystems. Technical report, Dynatrace LLC., January 2018. URL: https://info.dynatrace.com/global_all_whitepaper_cloud_complexity_report_11870_registration.html.

[10] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless Applications: Why, When,

and How? *IEEE Software*, 38(1):32–39, January 2021. Conference Name: IEEE Software. doi:10.1109/MS.2020.3023302.

[11] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, 3(5):10–14, September 2016. Conference Name: IEEE Cloud Computing. doi:10.1109/MCC.2016.105.

[12] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-Native Applications. *IEEE Cloud Computing*, 4(5):16–21, September 2017. doi:10.1109/MCC.2017.4250939.

[13] Tom Goethals, D. Kerkhove, Laurens Van Hoye, Merlijn Sebrechts, F. Turck, and B. Volckaert. FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers. In *CLOSER*, 2019. doi:10.5220/0007706000900099.

[14] Tom Goethals, Merlijn Sebrechts, Ankita Atrey, Bruno Volckaert, and Filip De Turck. Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 1–8, November 2018. doi:10.1109/SC2.2018.00008.

[15] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 98:27–42, November 2017. doi:10.1016/j.jnca.2017.09.002.

[16] Tom Laszewski, Kamal Arora, Erik Farr, and Piyum Zonooz. *Cloud Native Architectures: Design High-Availability and Cost-Effective Applications for the Cloud*. Packt Publishing, 2018.

[17] Dominique Lepore, Sabrina Dubbini, Alessandra Micozzi, and Francesca Spigarelli. Knowledge Sharing Opportunities for Industry 4.0 Firms. *Journal of the Knowledge Economy*, 13(1):501–520, March 2022. doi:10.1007/s13132-021-00750-9.

[18] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01*. 2013.

[19] István Pelle, János Czentye, János Dóka, and Balázs Sonkoly. Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 272–280, July 2019. doi:10.1109/CLOUD.2019.00054.

[20] Ramon Perez, Priscilla Benedetti, Matteo Pergolesi, Jaime Garcia-Reinoso, Aitor Zabala, Pablo Serrano, Mauro Femminella, Gianluca Reali, Kris Steenhaut, and Albert Banchs. Monitoring Platform Evolution Toward Serverless Computing for 5G and Beyond Systems. *IEEE Transactions on Network and Service Management*, 19(2):1489–1504, June 2022. Conference Name: IEEE Transactions on Network and Service Management. doi:10.1109/TNSM.2022.3150586.

[21] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, January 2021. doi:10.1016/j.future.2020.07.017.

[22] Merlijn Sebrechts, Tom Goethals, Thomas Dupont, Wannes Kerckhove, Ruben Taelman, Filip De Turck, and Bruno Volckaert. Solid Web Monetization. In *Proceedings of the 22st international conference on web engineering (ICWE 2022)*, Lecture Notes in Computer Science, Bari, 2022. Springer International Publishing. doi:10.1007/978-3-031-09917-5_40.

[23] Kenton Varda. Introducing Cloudflare Workers, September 2017. URL: http://blog.cloudflare.com/introducing-cloudflare-workers/.

[24] I. Wang, E. Liri, and K. K. Ramakrishnan. Supporting IoT Applications with Serverless Edge Clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–4, November 2020. doi:10.1109/CloudNet51028.2020.9335805.

[25] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.*, 50(2):26:1–26:41, May 2017. doi:10.1145/3054177.

[26] Chunyi Zhou, Anmin Fu, Shui Yu, Wei Yang, Huaqun Wang, and Yuqing Zhang. Privacy-Preserving Federated Learning in Fog Computing. *IEEE Internet of Things Journal*, 7(11):10782–10793, November 2020. doi:10.1109/JIOT.2020.2987958.

2

Orchestrator Conversation: Distributed Management of Cloud Applications

This chapter presents a published research article that tackles the first research question of this dissertation: "How to encapsulate and reuse system administrator's knowledge about when to deploy what?" Although this research was performed in 2017 and published in 2018, the question itself is still relevant to this day. Declarative desired state models are still the main way to manage cloud applications, even in state-of-the-art container orchestrators such as Kubernetes. Rightly so, since these models are very useful to describe the desired state of an application using the abstractions provided by the language. Nevertheless, they lack powerful tools to let users build new abstractions. This article proposes the "orchestrator conversation" to solve this issue. Although it uses declarative models as an API, it adds the concept of an "orchestration agent", which translates higher-level abstract models to lower-level concrete models. Benchmarks in the results section of this paper show varying standard deviation caused by variation in the response time of the AWS cloud infrastructure used during testing. Since these results represent real-world performance and predictability of cloud providers, we chose to include them as measured, instead of cleaning the data. Looking back at this work in 2022, however, there is an important piece of related work missing in this paper: Kubernetes Controllers and Custom Resources. Although this was still a beta feature released without much fanfare, it has now grown into an important cornerstone of Kubernetes. It is now the main way to create new abstractions for the Kubernetes API and is interestingly based on the same principle as an orchestration agent: using a regular programming language to translate higher-level abstractions into lower-level ones.

M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck

Published in International Journal of Network Management, July 2018.

Abstract Managing cloud applications is complex, and the current state of the art is not addressing this issue. The ever-growing software ecosystem continues to increase the knowledge required to manage cloud applications at a time when there is already an IT skills shortage. Solving this issue requires capturing IT operations knowledge in software so that this knowledge can be reused by sysadmins who do not have it. The presented research tackles this issue by introducing a new and fundamentally different way to approach cloud application management: a hierarchical collection of independent software agents, collectively managing the cloud application. Each agent encapsulates knowledge of how to manage specific parts of the cloud application, is driven by sending and receiving cloud models, and collaborates with other agents by communicating using conversations. The entirety of communication and collaboration in this collection is called the *orchestrator conversation*. A thorough evaluation shows the orchestrator conversation makes it possible to encapsulate IT operations knowledge that current solutions cannot, reduces the complexity of managing a cloud application and happens inherently concurrent. The evaluation also shows that the conversation figures out how to deploy a single big data cluster in less than 100 milliseconds, which scales linearly to less than 10 seconds for 100 clusters, resulting in a minimal overhead compared to the deployment time of at least 20 minutes with the state of the art.

2.1 Introduction

Managing cloud applications is complex. System Administrators (sysadmins) need to have an in-depth understanding of all the components of the cloud application such as the operating system, webserver, X.509 certificates and more. Having such deep knowledge about how to deploy, configure, monitor and manage these components is almost impossible in the field of big data because of the size of the ecosystem, the complexity of the tools involved and the rapid pace of innovation. This would not be such a big problem if it was not for the large skills shortage in the fields of IT operations [32] and big data [31]. There is thus a need for the ability to share and reuse the knowledge of sysadmins across teams and companies.

Sharing IT knowledge is not a new concept. The field of software development, for example, has a big focus on sharing and reusing knowledge in the form of code libraries. Over the years, a vast number of code libraries have been created that encapsulate an enormous amount of knowledge. Developers use these libraries to quickly write software without having to know each and every detail of how the software works. As an example, a programmer writing a piece of software that communicates over HTTPS does not need to know the intricate details of the TCP/IP protocol, X.509 certificates and SSL encryption. By simply using a library that implements all these functions, the programmer can focus on the actual novel parts of the application.

Two properties of programming languages are key enablers for this knowledge reuse: the ability to encapsulate code and to create new abstractions. **Encapsulation** allows developers to group code with a common function into a reusable module. By creating an **abstraction**, the developers expose the functionality of that module over an API that hides the inner complexity. An important property of these two is that they are stackable: a module can have varying levels of abstraction where each level encapsulates and hides the complexity of the level below it. As an example, a library for HTTPS communication might encapsulate three libraries: one for TCP/IP communication, one for X.509 certificate management and one for SSL encryption. Note that it is not sufficient for a programming language to be an abstraction of machine code, because it only allows developers to reuse the knowledge of the creators of the programming language, not the knowledge of other developers. It is vital the programming language itself allows for developers to create *new* abstractions *with* the language instead of *in* the language, so that they can easily encapsulate their own knowledge in a reusable way.

Knowledge reuse is regrettably a lot less prevalent in IT operations since it has historically been a mostly manual job which makes capturing knowledge hard. The rising popularity of configuration management automation provides new opportunities. Automation tools such as Chef and Puppet allow sysadmins to develop code that manages a cloud application. Although this code captures the sysadmin's knowledge, it does not enable knowledge reuse because building on top of automation code requires the same knowledge as creating the code in the first place. This issue stems from the foundational theory behind configuration management tools: converging towards a predefined end-state, as popularized by Burgess et al. [12] The idea of convergence is that a sysadmin specifies the desired end-state of an application and the configuration management tool executes the necessary actions to get the application into that state. The automation code is in that sense a description of the desired end state of the application. The same code thus always results in the same end state. Having the code figure out what the end state needs to be, is not possible in this convergent approach to configuration management. This has the advantage of consistency and reliability, but this presents a big issue for creating reusable automation code modules. When an automation code module changes a property of an application, there are two options. Either the module hides the value of that property in an abstraction, but then the value is static so the module cannot be used in a scenario where that property needs a different value, or the module exposes the property in its API, causing the module to leak its complexity. As a result, a module is either *understandable*, or *flexible*, but it cannot be both, even though both are important for successful reuse according to a systematic mapping study by Bombonatti et al. [7]

Note that it is entirely possible, and common, for the desired end-state to be an abstract description of the actual end-state. However, it becomes the responsibility of the configuration management tool to translate the abstract description into a concrete description. The only abstractions possible in configuration management tools are those provided by the creators of the tools. Sysadmins can thus only reuse the knowledge of the creators of the automation tools, not the knowledge of other sysadmins.

A recent addition to cloud application management is the concept of topology-based cloud mod-

eling languages such as the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA). On top of automation, these languages also provide ways to model the cloud application as a set of interdependent services, and they abstract the underlying cloud to prevent cloud vendor lock-in [41]. These languages are typically paired with an *orchestrator*, a program that interprets the model and performs the necessary management actions.

Reducing the complexity towards end-users, however, requires more than abstracting the cloud itself: it requires creating new abstractions using the cloud modeling language, which is still not possible. As an example, modeling and managing a Hadoop cluster as a single entity is not possible. System administrators need to model each individual component of a Hadoop cluster and their dependencies: the Namenode and Datanode to get an HDFS cluster, and the ResourceManager and NodeManager to get a YARN cluster, the Datanode and NodeManager must be co-located, etc. This also causes sysadmins to be responsible for translating higher-level objectives such as "scale the Hadoop cluster" into actions on the individual components of the cluster and requires them to have in-depth knowledge of the inner workings, i.e. on what it means to scale a Hadoop cluster.

Simply making it possible to create abstractions in cloud modeling languages is not the whole solution, however, because someone still needs to create these abstractions and encapsulate knowledge. The answer to the "who" question is complex because there are multiple parties involved in the creation and management of a cloud application:

- the system administrator that manages the cloud application,
- the Independent Software Vendor (ISV) of the software that makes up the cloud application,
- and the vendor of the orchestrator that interprets the cloud model.

The sysadmin is the expert on how the software is used in the cloud application, but it is the ISV who has the actual knowledge on how to manage the individual software artifacts, so the ISV is the prime candidate to encapsulate the knowledge. Note that the transfer of knowledge from ISV to sysadmins currently happens almost exclusively using documentation and tutorials. The ISV writes documentation and tutorials, and the sysadmin uses that documentation to manage the software. This means that the most important role of the cloud modeling language and orchestrator is to provide a platform to enable sysadmins to use the expertise of the ISVs. The orchestrator vendor cannot decide what the requirements are, since that is the role of the sysadmin, and the orchestrator vendor should not decide how individual components are best managed, since that is the role of the ISV. This is regrettably not the case with current cloud modeling languages. As an example, if a sysadmin uses the current declarative capabilities of the TOSCA cloud modeling language [30], then it is up to the orchestrator to interpret that model and manage the cloud application accordingly. This is the exact opposite of the ideal scenario, where the role of the orchestrator is as minimal as possible.

In summary, the issue with the current generation of cloud application management tools is that these tools do not enable sysadmins to reuse the knowledge of ISVs. The orchestrator conversation proposed in this article tackles this issue by introducing a new and fundamentally different way to approach cloud application management: the *orchestrator conversation*, a hierarchical collection of independent software agents that collectively manage the cloud application. Each agent is an orchestrator that manages a specific part of the cloud application, collaborates with other agents over peer-to-peer conversations and deploys new agents to delegate tasks to.

Three features of the orchestrator conversation are key to addressing the aforementioned issue.

a) The ability to encapsulate automation code that **manages a single service** and to provide that functionality over an abstract API makes it possible to reuse the knowledge of how to manage a single service.

b) Encapsulation of automation code that **orchestrates a number of services** and providing that functionality over an abstract API, which enables reusing the knowledge of how to orchestrate multiple interconnected services.

c) The orchestrator conversation enables multiple independent ISVs to encapsulate their knowledge **in an interoperable way** so that sysadmins can build a cloud application using mul-

tiple components from multiple ISVs.

The remainder of this article is structured as follows. Section 2.2 gives an overview of the state of the art concerning this field and identifies how the related work has influenced this article. Section 2.3 explains the concepts behind the orchestrator conversation, and how they address the identified issues. Section 2.4 evaluates the solution using simulations and Section 2.5 concludes this article.

2.2 Related work

There is a lot of different terminology being used regarding the management of cloud applications. Since these terms are not always used in a consistent manner, this section starts with a description of how these terms are used in the context of the presented research. This generally follows the cloud resource orchestration taxonomy proposed by Weerasiri et al. [45]

A *cloud application* consists of a number of *services* connected by *relationships*. A relationship generally denotes a dependency and/or an interaction between two services. An *application topology* is a description wherein the cloud application is described as a graph of nodes (the services), and edges (the relationships). *Cloud resource orchestration* is the process of ***selecting***, ***describing***, ***deploying***, ***configuring***, ***monitoring*** and ***controlling*** the infrastructure and services that make up the cloud application. Cloud resource orchestration is abbreviated as *orchestration* and is interchangeable with *cloud application management* in the context of this research. An orchestrator is a piece of software that performs orchestration tasks. [1]

A number of efforts from different fields try to tackle the issues regarding management of cloud applications. The remainder of this section explores how each field addresses different issues, how this relates to knowledge reuse, abstraction and collaboration, what our research takes away from these efforts, and how it goes beyond the state of the art.

2.2.1 Resource scheduling

The lines between cloud application management and resource scheduling are starting to blur, resulting in a number of innovative solutions regarding cloud application management to come out of the resource scheduling field. The reason for this evolution is that there is a lot more to the management of a cluster than simple job scheduling. Jobs are part of larger applications that have complex topologies and dependencies. Jobs have a complex lifecycle, they need to communicate with each other and they need to be configured [43].

One of the big lessons learned from this field is that monolithic schedulers evolve into complex hard-to-maintain systems because of the increasing heterogeneity of resources and jobs and the widening range of requirements and policies. This problem is addressed by pulling the monolithic

[1] Note that "orchestration" does not imply a central controlling entity. There is thus no distinction between orchestration and choreography in this context.

cluster manager apart into a number of specialized schedulers that work together on shared resources and job queues [34]. Apart from solving the complexity issue, this approach also makes it possible to have multiple schedulers from different vendors manage shared resources [24].

The big shortcomings of the state of the art in cluster management are the lack of native support for grouping workloads into application topologies [43], automatic dependency management and dynamic reconfiguration of workloads [13]. The need for dynamic reconfiguration is inherent to configuration itself since configuration solely exists to make hard-coded parameters changeable. If a parameter needs the same value for every deployment of an application, then this parameter will simply be hard-coded in the application's source code. The advent of microservices has only increased the complexity of application topologies and their configurations and dependencies, making it impossible to manage these manually [23].

2.2.2 Cloud modeling languages

Cloud modeling languages provide a standardized format to ***describe*** cloud applications and their metadata. The sysadmin creates a model that describes the desired state of the cloud application and the orchestrator ***deploys*** and ***configures*** the cloud application according to that description. These languages thus have enormous potential to encapsulate cloud application management knowledge in a reusable manner. The latest generation of cloud modeling languages is centered around describing the application as a topology of components and their relationships to each other. The TOSCA language is an effort to reduce vendor lock-in by separating the cloud modeling language from the cloud provider and cloud infrastructure platform. This effort resulted in a push towards abstraction in cloud modeling languages. Abstraction of individual components is possible using TOSCA "node templates". The orchestrator substitutes a node template by concrete node types before deployment [30]. Brogi et al. identified four different matching strategies for transforming these abstract node templates into concrete node types [11]. These strategies can be used to combat vendor lock-in by using standardized vendor-neutral node-templates [10]. The downside is that this substitution is a one-way process which results in critical information loss. The resulting topology does not contain any information about what node templates were present nor what node types correspond to what node templates. As a result, a sysadmin can only use node templates during the deployment phase. After the deployment is done, the abstractions are lost and the sysadmin is exposed to the full complexity of the cloud application making the monitoring and controlling phases very complex.

A useful feature of the TOSCA approach is that it is possible to create a topology that contains both node types and node templates, thus having multiple levels of abstraction in a single model. This allows sysadmins to choose the appropriate level of abstraction for each part of the topology. As an example, a sysadmin can use the "SQL Database" node template for parts of the cloud application that are database agnostic and use the "MariaDB Database" node type, in the same model, for parts of the cloud application that are tied to that specific database.

A big advantage of topology-based cloud modeling languages is that each individual component is

isolated. Interaction between components is only possible using relationships. This makes it very easy to create reusable components, resulting in community-driven capturing and reuse of configuration management code [42][21][46], which is very important in the field of cloud resource orchestration [45]. However, the actual reuse of knowledge is limited because of the issues explained in the introduction.

The following research efforts have made progress towards formalizing the concepts behind topology-based cloud modeling languages. Andrikopoulos et al. propose a set of formal definitions to reason on topology-based cloud applications with the goal of ***selecting*** the optimal distribution [2]. The Aeolus component model makes it possible to formally describe several characteristics of a cloud topology such as dependencies, conflicts, non-functional requirements and internal component state [19]. Brogi et al. propose a Petri net-based approach to formally model the relationships of TOSCA cloud models [9].

A big shortcoming of cloud modeling languages is the limited support for creating abstractions using cloud models. This has caused TOSCA orchestrators such as Cloudify to build their own methods to enable this. Cloudify's solution to this is Cloudify Plugins [18]. A Cloudify plugin basically allows adding additional orchestration logic into the Cloudify Orchestrator. These plugins allow to define new base node types and control how the orchestrator handles them. Although this method provides a way for Cloudify users to create new abstractions, it is still lacking. For instance, this method does not make it possible to stack abstractions: you cannot create new abstractions by combining and encapsulating a number of existing abstractions. Furthermore, this method nullifies an important property of TOSCA models: their portability. A TOSCA model that uses a plugin-specific node type cannot be interpreted by an orchestrator that does not support the specific plugin. Since plugins are developed using a Cloudify-specific API, this essentially reintroduces vendor lock-in into the TOSCA ecosystem. This is a big issue, given that TOSCA's main selling point is that it eliminates vendor lock-in. Note that this method of extending orchestrators to enable abstraction is neither limited to Cloudify nor to TOSCA. The alien4cloud project [4] provides an abstraction layer on top of TOSCA models, and the conjure-up project [15] provides an abstraction layer on top of the Juju [16] cloud modeling language discussed in Section 2.2.4.

2.2.3 Models at runtime

Cloud modeling languages can also be used to ***monitor*** and ***control*** runtime state [6]. The *models at runtime* (M@RT) approach is to have a model that is causally connected to the running application: the runtime model is constantly updated to mirror the runtime state [39][38]. Another approach called *self-modeling* [25] is to dynamically generate a model from the current state using generic building blocks, which has been shown to be useful for self-diagnosis and root-cause analysis [40]. These approaches, however, are limited in that they only support a one to one mapping between the runtime model and the runtime state. This is an issue because complex abstractions can violate this constraint: a single abstraction can represent different runtime states in different topologies and different abstractions can represent the same runtime state in different

topologies, as is the case with the "Spark on Hadoop" example used for evaluation in Section 2.4. Solving this issue requires more complex translations between the running application and the model than the M@RT approach currently provides.

2.2.4 Agent-based management of cloud applications

The approach of converging towards a predefined end-state as popularized by Burgess et al. [12] is inherently inflexible as explained in the introduction. We believe that agent-based cloud management addresses this inflexibility. An agent-based cloud application manager consists of a number of independent agents that each manage a specific part of the cloud application i.e. a service. Each agent independently decides what the desired end-state is for that service and executes the necessary actions to get into that state. Dependencies between services are resolved by communication between the agents. As an example, a cloud application consisting of two services, a website and a database, is managed by two agents. One agent is responsible for the website and the other one for the database. If the website needs a connection to a running database, the agent responsible for the website will wait until it receives a message from the agent responsible for the database saying that the database is running. In this approach to config management, the global end-state of the cloud application emerges from the collective behavior of the agents.

One of the big advantages of an agent-based approach is the reliability and resiliency against failures as shown by Xavier et al. [22] and Kirschnick et al. [26]. Lavinal et al. have shown that the local autonomy of each agent combined with their organizing behavior enables global management autonomy in a distributed environment [28]. The flexibility of the agent-based approach allows it to manage not only analytical platforms but also the workloads running on top of those platforms as shown by the authors' previous work [36]. The state of the art in this area however does not address the need for abstraction, collaboration and reuse.

Juju [16] is a cloud modeling language and orchestrator that can be seen as a hybrid between agent-based management and cloud modeling languages. The sysadmin creates a Juju model describing the entire topology of the cloud application, the Juju orchestrator interprets that model, but the actual management of the individual services is done by agents co-located with the services. Juju makes it possible to encapsulate automation code that manages a single service in a charm, an entity similar to a TOSCA node type, but charms are not stackable and can only manage a single service.

2.2.5 General limitations and lessons learned

A recent survey on cloud orchestration by Weerasiri et al. [45] identifies a number of general limitations across the field of cloud application management. Although concerns such as conformance to QoS and SLA requirements are reasonably addressed by the state of the art, the importance of knowledge reuse is underestimated and there is too much fragmentation resulting in sysadmins having to use different tools to manage different aspects of the application lifecycle [45]. The survey furthermore proposed the idea of *orchestration knowledge graphs*, where *"common low-level*

orchestration logic can be abstracted, incrementally curated and thereby reused by DevOps" [45].

For this reason, the presented research's focus is not in finding new orchestration and scheduling techniques, but in developing a specification that makes it possible for the existing cloud schedulers, modeling languages and orchestrators to work together, enabling encapsulation of cloud application management knowledge in orchestration knowledge graphs.

There are a number of concepts in the state of the art that are very useful in achieving this. The presented research uses the following concepts as a foundation for the orchestrator conversation as explained in the next section.

a) The concept of a meta-scheduler as a way for different schedulers to work together solves real problems and should be applied to orchestration as well.

b) Topology-based cloud modeling languages make it easy to represent and reason over a cloud application and enable communities to collaborate around encapsulated knowledge.

c) Runtime models are a great way to represent the current state of a cloud application.

d) The agent-based approach to deployment of cloud applications provides a lot of benefits to the resiliency of the management system.

2.3 Orchestrator conversation

We propose the orchestrator conversation as a fundamental new way to approach cloud modeling languages and orchestrators. This section gradually introduces the key concepts behind the orchestrator conversation. As a running example, the management of a Hadoop cluster is used. The section concludes with a summary of the entire conversation.

The need for knowledge reuse has heavily influenced the design decisions in this section. Specifically, the systematic mapping study of software reuse by Bombonatti et al. is used as a guideline for the non-functional requirements of the orchestrator conversation: *understandability*, *modularity* with *loose coupling*, *flexibility* and *abstractness*.

2.3.1 Request and runtime models

Two pieces of information are key to the management of cloud applications: what state should the application be in, and what state is the application currently in. In the orchestrator conversation, that information is embedded in two types of cloud models.

Definition 2.1. *The **request model** is a structured description of the desired state of part of the cloud application.*

Definition 2.2. *The **runtime model** is a structured description of the actual state of part of the cloud application at a specific point in time.*

These models follow a predefined schema that is both human- and machine-readable so both system administrators and the management platform itself can *understand* them. Cloud modeling languages are perfectly fit as the schema for these models, since they provide a way to describe cloud applications satisfying both constraints. The request and response models are always linked in the sense that the runtime model describes the state in the context of the request model. It must be clear from the runtime model what the status is of the requests in the request model. As the example in Figure 2.1 shows, the runtime model uses the same names for the service and its properties so that the sysadmin understands the runtime model.

These two models are the primary way for a system administrator to communicate with the management platform. The sysadmin creates a request model to tell the management platform what the desired state of the cloud application is and uses the runtime model to keep track of the runtime state of the cloud application.

Capturing the complete and current state of a highly distributed cloud application requires strict consistency, which degrades performance immensely because of global locks. Furthermore, locking the state of a running cloud application without any downtime is arguably impossible since state changes such as a running process crashing unexpectedly cannot be prevented. This is the reason why the runtime model does not represent the state of the cloud application in the present, but at some time in the past. The orchestrator conversation follows the eventual consistency model [44]: if the state does not change, the runtime model will eventually reflect the current state. This is achieved by ensuring the following three properties.

- The runtime model represents the state at some point in the past.
- If the state has changed since that point, the runtime model will be updated at some point in the future.
- Write conflicts are handled using the “last writer wins” approach.

2.3.2 Service Agent

The actual management of the individual services is the responsibility of the service agent (SA).

Definition 2.3. *A **service agent** is an event-based program that manages a single service to get it into the state described in the request model and creates the runtime model that reflects the runtime state of the service.*

A service agent is fully event-based, thus it only reacts to changes, which can be internal such as a service crash, or external such as an update to the request model. Each change generates an event that the service agent needs to process. Each service agent has a specialized role. For example, Figure 2.1 shows a service agent that is specialized in managing a Hadoop Worker service. The service agent is running on the same machine as the service itself. This is an easy way to give the service agent access to the service, although this is not required as shown in the authors' previous work [37].

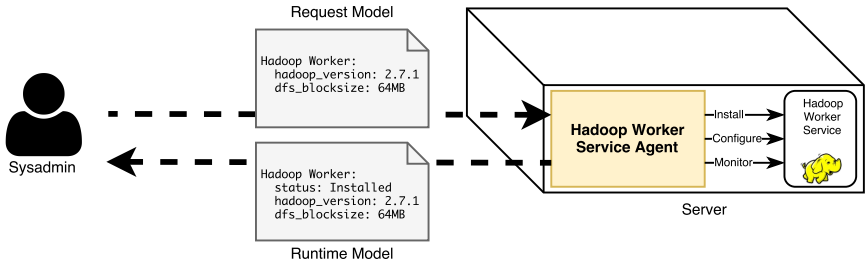

Figure 2.1: The Hadoop Worker service agent runs on the same machine as the Hadoop Worker service. The sysadmin defines the desired state of the service in the request model and the service agent notifies the sysadmin of the current state using the runtime model.

In the scenario shown in Figure 2.1, the process is the following.

1. The system administrator **selects which service agent to use.**
2. The sysadmin deploys the service agent onto a server.
3. The sysadmin sends the request model to the service agent and subscribes to its runtime model.
4. The service agent deploys and configures the service to get it into the state described in the request model.
5. The service agent updates the runtime model to reflect the current state and sends it to the sysadmin.

A key difference between the service agent approach and conventional configuration management tools is that the system administrator not only chooses the end state, but also the entity that will interpret that end state, i.e. the service agent. This solves the issue explained in the introduction that the languages used to describe the end state do not support creating flexible *abstractions*. Service agents provide that functionality and can contain arbitrary processing logic to translate an abstract request model into a practical set of operation actions that need to happen. As a result, sysadmins can encapsulate knowledge in a service agent so it can be reused. Figure 2.1 shows an example where the request and response models only specify a few configuration options. The service agent decides what the optimal values are for the unspecified configuration options. As a result, *flexibility* and *understandability* are not mutually-exclusive in this approach. Aside from enabling knowledge reuse, the service agent also has a number of other advantages:

- The need for a one-size-fits-all cloud modeling language is removed because different languages can be used for management of different services. As an example, modeling a Virtual Network Function (VNF) is quite different from modeling a big data service so it is useful to model each in a modeling language specific to its domain.

- Service agents allow for fine-tuned translations from declarative request models into imperative steps instead of having to rely on generic translation rules provided by the management software.
- System administrators can reuse existing "legacy" models such as a TOSCA topology in the orchestrator conversation. This allows them to capitalize on their existing investments in model-driven management of cloud applications and it provides an easy migration path from conventional management tools.
- The approach enables a heterogeneous ecosystem where cloud modeling languages can compete with each other and evolve quickly.

2.3.3 Collaborator relationship

The previous subsection focused on managing a single service. However, cloud applications generally consist of multiple interconnected services. A standard Hadoop setup for example requires three services working together: the Namenode, the ResourceManager and the Hadoop Worker. Connecting these services requires communication between the respective service agents in order to exchange information such as IP addresses and port numbers. The collaborator relationship enables this communication between service agents.

Definition 2.4. *A **collaborator relationship** is an isolated, two-way communication channel that connects two service agents and allows a **conversation** between them. Each service agent has a **role** in this conversation, which denotes how the service agent acts. Two types of conversations are possible: unary and binary. In a **unary** conversation, both service agents have the same role. In a **binary** conversation, each service agent has a different role. A collaborator relationship between two service agents is possible if either both implement the same role of a peer-to-peer conversation or if they both implement opposite roles of a directional conversation.*

The term *conversation* is chosen to differentiate from the simple static exchange of properties possible in languages such as TOSCA. Similar to its meaning in Business Process Model and Notation (BPMN), a conversation can consist of multiple interactions and messages resulting in negotiation and resolution of complex dependencies. The Hadoop cluster shown in Figure 2.2 shows three directional conversations. In each conversation, both connected service agents have a distinct role. For example, the Hadoop Namenode and the Hadoop ResourceManager have two distinct roles in their conversation simply because they provide information about two distinct services. The collaborator relationships in this example are used to check Hadoop version compatibility, exchange IP addresses and port numbers, setup shared credentials, and coordinate service starts and restarts. A sysadmin creates a relationship by specifying a request for the relationship in the request models of both service agents. Each relationship request contains the address of the other service agent, the roles of both service agents and the conversation protocol.

Note that a relationship only denotes that two agents can communicate. Whether or not that communication will be fruitful is not specified. As an example, during the conversation between the

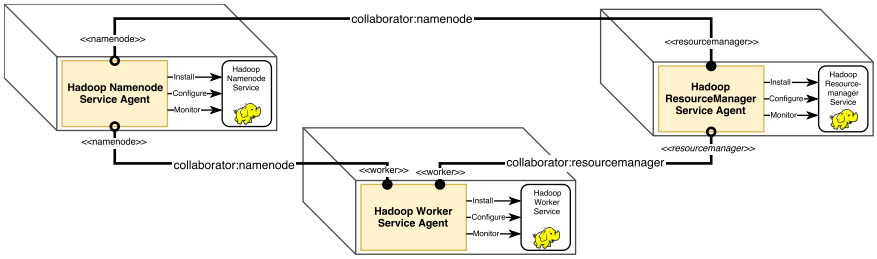

Figure 2.2: Illustrative example: the collaborator relationships between SA's of a Hadoop cluster.

Hadoop Namenode and the Hadoop ResourceManager, it might become clear that the ResourceManager's Hadoop version is incompatible with the Namenode's Hadoop version. If the Hadoop version is specified in the request model, then the service agents are not allowed to change the Hadoop version themselves. It is then the responsibility of both service agents to explain this issue in their runtime models so that the sysadmin can intervene, e.g. by changing the ResourceManager's Hadoop version.

The collaborator relationship is a key piece to enabling collaboration because it allows multiple parties to develop service agents independently, while still allowing these service agents to collaborate and communicate. A developer can implement the collaborator conversation without knowing the implementation details of the service agent on the other side because the conversation acts as a generic protocol that hides the implementation. As a result, ISVs can create service agents that manage their own software and connect to software from other vendors. Orchestrator vendors facilitate this collaboration by creating a set of standardized collaborator conversation protocols that ISVs can program against. This will result in a rich ecosystem of interoperable service agents at the fingertips of system administrators who use them as building blocks for their cloud application.

2.3.4 Controller

Deploying multiple service agents introduces a new issue: the system administrator needs to create a request model for each service agent, which would be very cumbersome to do manually. This is where the topology-based cloud modeling languages come in. Such languages provide a way to describe a cloud application as a set of interconnected services. They are thus ideal for creating the combined request model including the relationship request between the different service agents. Dividing the topology model into a number of request models is quite straightforward: each vertex is a separate request model. Each edge is described in both request models of the nodes it connects. Doing this division is the job of the *controller*. The controller receives the entire request topology model, divides the request model and sends each part to the service agent responsible for that service.

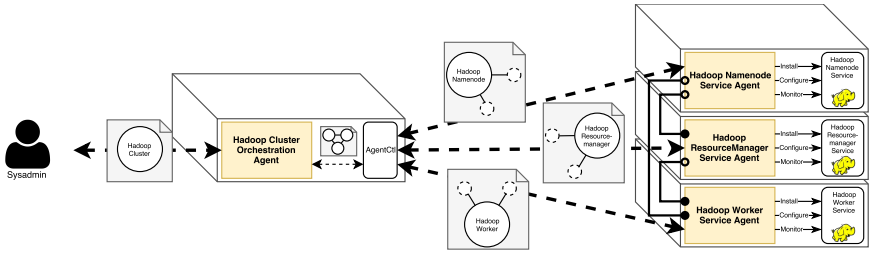

Figure 2.3: The Hadoop cluster orchestration agent manages multiple service agents by sending them individual request models.

2.3.5 Operator relationship

The service agent as described in the previous sections enables abstraction of one single service, but it does not allow abstracting an entire topology into a single component. As an example, when setting up the Hadoop cluster, the sysadmin still needs to know that a Hadoop cluster consists of a Namenode, Datanode and a Worker, and how they need to be connected. As explained in the introduction, there is a need for an abstraction layer that can represent a cluster of services as one service, so the system administrator can request a deployment of "a Hadoop cluster" and have the management platform figure out what services are required for a Hadoop cluster. As shown in the state of the art section, it is important that this abstraction is two-way: both the request and response model the sysadmin interacts with need to represent the individual Hadoop services as one Hadoop cluster.

Figure 2.3 shows the orchestrator conversation's solution to this: a service agent that takes over the job of the system administrator. This service agent receives the abstract request for the Hadoop cluster and fulfills that request by creating new service agents, sending them request models, listening for runtime models, translating those runtime models into one overall state of the Hadoop cluster, and sending a runtime model reflecting that state to the sysadmin. The key to enabling this is to characterize the interaction between the sysadmin and a service agent and create the operator relationship that allows such interaction between service agents themselves.

Definition 2.5. *An **operator relationship** is an isolated, two-way communication channel between two service agents that allows one service agent to send request models to and receive response models from the other service agent. The conversation going over the operator relationship has two roles: the **operator** sends request models to drive the behavior of the **executor**, which sends runtime models back to the operator to inform the operator about the runtime state. A service agent can be the executor in only one operator relationship, but it can be the operator in multiple relationships.*

This article refers to a service agent that manages a number of other service agents using an operator relationship as an "**orchestration agent (OA)**". The interaction between a system administrator and an OA can be seen as the system administrator having an operator relationship to

the service agent. In fact, there is no practical difference between a system administrator managing a service agent or another service agent managing that service agent. As a result, it is possible to create multiple layers of abstraction by chaining service agents using operator relationships.

This does have implications on the visibility of the cluster of service agents to the system administrator. Only the service agents that are directly connected to the sysadmin are fully visible, which is wanted behavior since it enables abstraction: the visibility of the service agents in lower abstraction layers is curated by the service agents in the upper abstraction layers. Subsequently, each service agent has complete control over its executors.

The operator relationship is the second key piece for enabling a rich ecosystem of orchestration knowledge. Just as the collaborator relationship, the operator relationship serves as a generic protocol that service agent developers can program against. ISVs can now create service agents that manage entire clusters of their software and orchestrator vendors can create service agents that translate higher-level requests to lower-level setups.

2.3.6 Summary

The orchestrator conversation consists of a hierarchical collection of *service agents* that collaborate to deploy a cloud application. Each service agent is an orchestrator specialized in managing a specific part of the cloud application. A sysadmin deploys service agents and sends them *request models* to specify what the desired end-state is. These service agents then communicate using *collaborator relationships* to connect different parts of the cloud application and delegate work by deploying new service agents and managing them using the *operator relationship*. The service agents report back to the sysadmin using eventually-consistent *runtime models* that show the state of the cloud application in the context of the request model.

Service agents encapsulate orchestration logic and expose their functionality over abstract APIs, thus enabling knowledge reuse. Relationships between service agents use agreed-upon conversation protocols, enabling a vibrant ecosystem of multiple vendors creating interoperable service agents. Orchestration vendors can ease this collaboration by standardizing conversation protocols and providing generic orchestration agents that perform tasks such as auto-scaling. The end-result is a *modular* system with *loosely coupled* components.

The orchestrator conversation can be seen as a swarm in the sense that it consists of a number of locally interacting SAs that collectively achieve the goal of managing the cloud application without a centralized control structure. An SA's interactions are strictly local because communication can only happen over relationships, the local neighborhood of a SA is determined by its relations and the management of the cloud application is an emergent behavior since no single SA has the knowledge to manage the entire cloud application.

2.3.7 Aside: declarative and imperative modeling

The topic of declarative versus imperative modeling is subject of an ongoing debate in the field of cloud modeling and orchestration [8][27][20]. This also introduces the issue of uncertainty since different orchestrators can interpret a model in different ways, causing the orchestration to fail in unexpected ways. For this reason, ISVs have a strong preference for imperative models.

ISVs see cloud modeling languages as a way to enhance the experience of their customers by accompanying their software with a model that encapsulates the knowledge of how to deploy and manage the individual software components. Imperative models allow them to fine-tune the models and have more control over the quality of experience of their customer, regardless of what orchestrator the customer uses.

The abstraction capabilities introduced by the orchestrator conversation are a great way to have the best of both worlds. The ISVs create the orchestration agents in an imperative way so they have full control over what the orchestration actions are and the customer interacts with the orchestration agent using declarative request models. The ISVs have full control over the orchestration actions and in turn over the quality of experience of their customers, while the complexity is still hidden to the user.

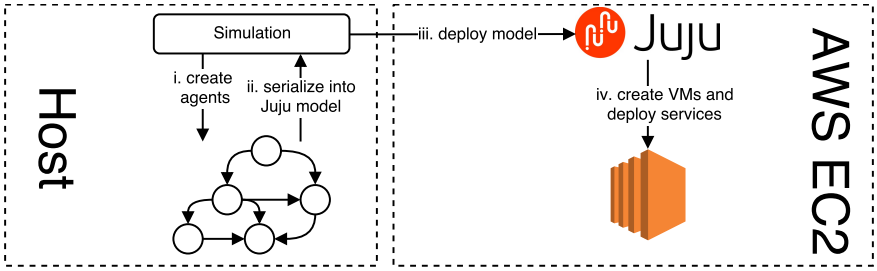

Figure 2.4: The evaluation setup. The orchestrator conversation is simulated on the host. The resulting topology is serialized into a Juju model which is deployed on AWS EC2.

2.4 Evaluation

2.4.1 Evaluation Setup

To evaluate the proposed orchestrator conversation, a number of proof-of-concept orchestration agents and simulated service agents are created using Python and the "Pykka" actor model framework. The full source code is available on Github[35]. The service agents are simulated in the sense that they do not actually deploy and manage services, they only track what state these services need to be in. When the orchestrator conversation finishes, the agents recursively serialize into a Juju bundle that describes the desired cloud application state. This bundle is then used to actually deploy the cloud application as shown in Figure 2.4.

All benchmarks of the orchestrator conversation were run on an Ubuntu 17.10 host with a 4-core Intel i5-7440HQ CPU and 16 GB of RAM. Deployment benchmarks were run using Juju 2.2.5 on AWS EC2 virtual machines of type "m3.medium" running Ubuntu 16.04 with 1 vCPU and 3.840 GB of RAM. The numbers of runs of the experiments and simulations are chosen so that there is sufficient convergence in the results and the standard deviation is small enough to show the significance of relevant trends.

2.4.2 Complexity towards the sysadmin

The first evaluation checks whether the orchestrator conversation makes it possible to hide the complexity of managing a cloud application to the sysadmin. The primary way of interfacing with a model-based cloud management platform is the model itself. Thus, the complexity of the model that the sysadmin interfaces with, is a good approximation of the complexity of using that management platform.

We compare the request model for a Hadoop cluster in the orchestrator conversation simulation with three models from the following state of the art projects: DICE, INDIGO DataCloud and Apache Bigtop. The exact models used are available on Github [35].

- DICE is a European Horizon 2020 project aimed at defining a framework for quality-driven

development of Big Data applications [17], which leverages TOSCA models to deploy and manage big data applications. This evaluation uses their example Hadoop models [1].

- The INDIGO - DataCloud project develops an open source data and computing platform targeted at scientific communities [33]. TOSCA is used as the method to interface between the INDIGO platform and end users. This evaluation uses the example model to request a Hadoop cluster from the INDIGO platform [29].

- Apache Bigtop is an Apache Foundation project that is the de-facto standard for packaging, deployment and testing tools for open-source big data frameworks [3]. It is the upstream of many commercial big data offerings such as Cloudera's CDH Hadoop distribution. This evaluation uses the Bigtop reference bundle for deploying Apache Hadoop using Ubuntu Juju [14][5].

This evaluation uses four indicators of the complexity of a model. Each object is a variable that needs to be specified by the user, thus increasing the complexity of the model. Moreover, each individual object acts as a multiplier to the overall complexity because the values of different objects need to be correct in combination. The number of objects can thus be seen as the degrees of freedom of the model. The indicators are as follows.

1. The number of **nodes** in the topology model. This is the number of declared node types for a TOSCA model and the number of declared applications and machines for a Juju model. Node types referenced but not defined are not counted since defining these node types is the role of the platform and thus provides no additional complexity for the sysadmin.

2. The number of **relationships** in the topology model. Each relationship is only counted once, even if it is declared at both endpoints such as in certain TOSCA models. Machine placement directives in Juju models are also counted as relationships since these signify a relationship between the application and the machine.

3. The number of **outputs**. In TOSCA models, the request model defines what runtime properties the sysadmin is interested in, and how these runtime properties should be formatted. Juju request models do not contain outputs because it is up to the Charm to decide what information should be shown to the sysadmin and how it should be formatted.

4. Number of **properties** present in the model. This maps to the number of properties in the TOSCA models, and the number of configuration values, constraints, and scale declarations in the Juju models.

Figure 2.5 compares the complexity of the four evaluated models and Figure 2.6 shows the topologies of these models as a graph. The indicators diverge a lot between the different state-of-the-art models because each model makes a different trade-off between flexibility and complexity: more information exposed in the model means greater flexibility but also more complexity. This trade-off is inherent to the state-of-the-art, since it is not possible to have both, as explained in the introduction. The trade-offs of the presented models are further investigated below.

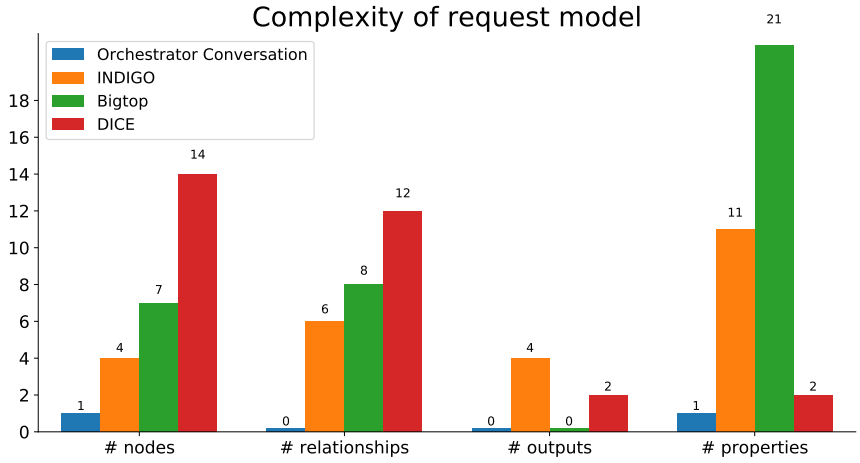

Figure 2.5: The request model for the orchestrator conversation only contains one node, the Hadoop cluster node, and one property, the scale of the Hadoop cluster.

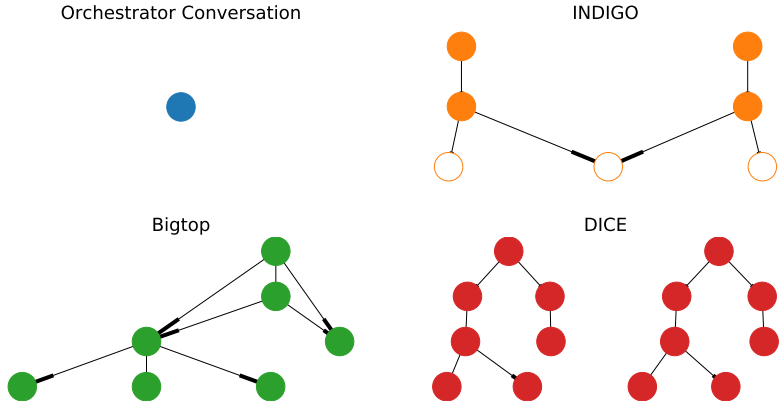

Figure 2.6: The request models of a Hadoop cluster in the different formats. The unfilled circles in the INDIGO model represent node types that are referenced in relations but not defined in the model itself.

- The DICE model as a large number of nodes and relationships because the IP, firewall, and virtual machine are also modeled as separate nodes with relationships. Although this is TOSCA-compliant and improves the reusability of the nodes, it greatly increases the complexity of the model.
- In the INDIGO model, the Hadoop services are only represented by two services: "hadoop_master" and "hadoop_slave" instead of the four separate services "Namenode", "Datanode", "NodeManager" and "ResourceManager". This causes the INDIGO model to be less complex, but it hampers the flexibility and reusability of the components in the model.
- The Bigtop and INDIGO models contains a large number of properties because the requirements of the host machines such as CPU, RAM and root disk space are part of the model. These properties are not set in the other models, which will result in the orchestrator deciding these values, causing the user to have no control over this..

The request model for the orchestrator conversation scores the best on every indicator which proves that this approach indeed makes it possible to hide the complexity of managing a Hadoop cluster. On the other hand, the sysadmin can still choose to manually model the Hadoop cluster out of individual components, or use a mix of components with different abstraction levels, should that need arise. Thus, the trade-offs between complexity and flexibility are not needed in the model of the orchestrator conversation.

Note that, as explained in Section 2.2, it is technically possible to make the TOSCA models easier using Cloudify plugins or node templates, but these methods have great limitations: node templates do not exist after deployment of the model and Cloudify plugins are neither standards-based nor stackable. In contrast, in the orchestrator conversation, the sysadmin can still manage the cloud application at runtime using the "Hadoop Cluster" abstraction, and the abstraction is completely stackable, the underlying individual service agents and their request models are still present.

2.4.3 Overhead of the orchestrator conversation

Some studies report that the reusability of software has a negative impact on its performance due to the overhead of abstraction and the absence of context-specific optimizations [7]. In order to see whether this is true for the orchestrator conversation, this series of evaluations investigates the overhead of turning an abstract request model into a full topology that satisfies it.

Since the main interest is in the overhead of the abstraction itself, the orchestrator conversation is only used to figure out what needs to be deployed in order to satisfy the initial request model. The orchestration agents create, configure and connect the required service agents but these do not actually deploy or manage any services, they simply figure out the desired state of the service and immediately report in their runtime model that the request model has been satisfied, which propagates through the topology until the initial request model is completely satisfied. At this point, the simulation stops and the desired state of all services is serialized into a Juju topology model. This model is then used to confirm if the required state as described by the service agents

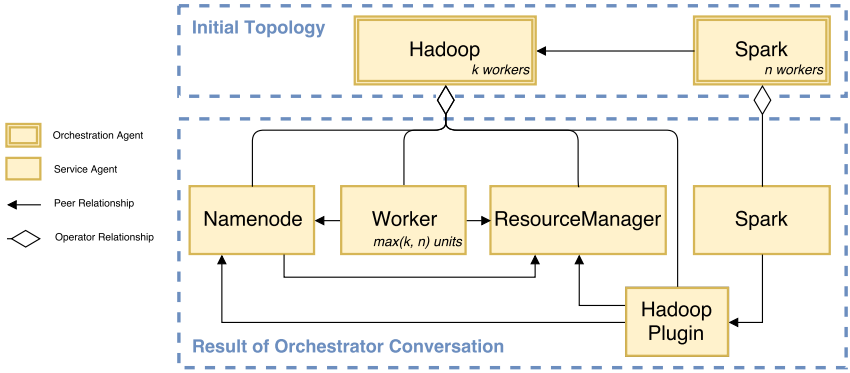

Figure 2.7: The simulation starts with a Hadoop OA and a Spark OA connected to each other. The goal is to create a Spark cluster running on a Hadoop cluster that consists of a Namenode, a ResourceManager and a Worker.

is correct, and deployed using Juju to get the time required to deploy the cluster using a state-of-the-art orchestrator.

The initial request model for this simulation is shown in Figure 2.7 as the "initial topology": a Hadoop orchestration agent connected to a Spark orchestration agent. The goal is to create the complete topology of orchestration and service agents which satisfies the request model. The request model of each agent contains a requested number of workers, referred to as k for Hadoop and n for Spark. During the simulation, the orchestration agents create new orchestration agents, service agents and relationships in order to achieve the desired state of the request model. The simulation finishes when all orchestration agents notify the user that the request model has been satisfied, thus when the complete topology is created. During the simulation, the two orchestration agents use the relationship to figure out what other agents need to be created and what the desired state is to fulfill the request. What follows is a summary of the actions and decisions that have to be made by the orchestration agents in this simulation.

1. The Hadoop OA creates service agents that deploy and manage a Hadoop cluster.
2. The Hadoop OA also creates the Hadoop plugin service agent. This plugin SA provides its peers with the correct information to connect an application to Hadoop. The Hadoop OA then sends the address of the plugin SA to the Spark OA so that the Spark OA can create a relationship between the plugin SA and the Spark SA.
3. The Spark OA creates a single Spark Client service agent and connects it to the plugin SA. Since Spark has to run on Hadoop, there is no need to create a full Spark cluster.
4. The Spark OA requests the Hadoop OA to create n workers since each Hadoop worker functions as a Spark worker when Spark runs on Hadoop.
5. The Hadoop OA compares n to k, the amount of workers from its request model, and up-

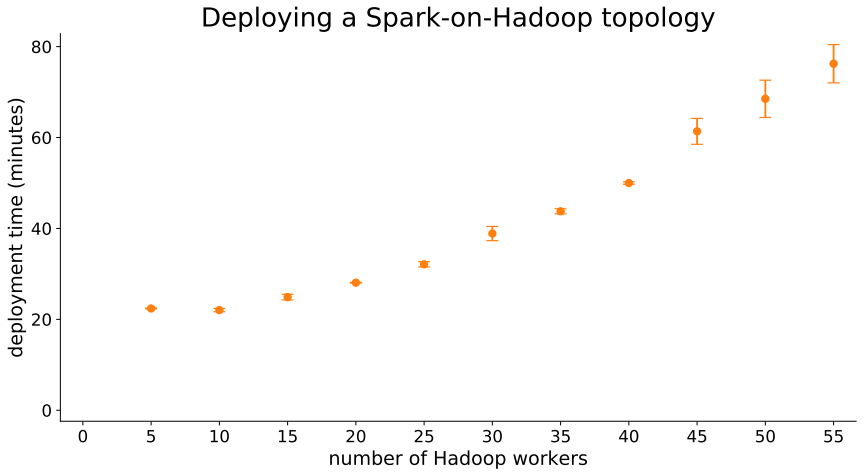

Figure 2.8: The minimum deployment time of the cluster is around 20 minutes and scales up with the number of workers.

dates the request model of the worker SA to create *max(n,k)* workers.

This interaction clearly shows the strength and flexibility of the orchestrator conversation. The abstraction of the Spark OA cannot be provided by a TOSCA node template because there is no one-to-one translation from the abstract "Spark cluster" node to what will actually be deployed: if the Spark OA is connected to a Hadoop OA, it will only deploy a single Spark client, otherwise it will deploy a full Spark standalone cluster. Similarly, the conventional M@RT approach will not work here because there is no one-to-one mapping between what the Spark OA reports as "number of Spark workers" and what is deployed: when it is connected to the Hadoop OA, the Spark OA will report the number of Hadoop workers, otherwise it will report the scale of the Spark standalone cluster. After the simulation finishes, all service agents serialize the desired state of the service into a Juju bundle, which is deployed to get the deployment time of the cloud application.

Figure 2.9 shows the time required by the orchestrator conversation to create the complete topology, starting from the request model. Each dot represents the aggregated result of 10 simulations of the orchestrator conversation. Figure 2.8 shows the deployment time: the time required by Juju to deploy the topology model created by the orchestrator conversation simulation. Each dot represents the aggregated results of two deployments. While the simulation time stays under 100ms, the deployment time ranges from around 20 minutes to over one hour. This demonstrates that the overhead of using orchestration agents as an abstraction is negligible compared to the deployment time of the actual cloud application.

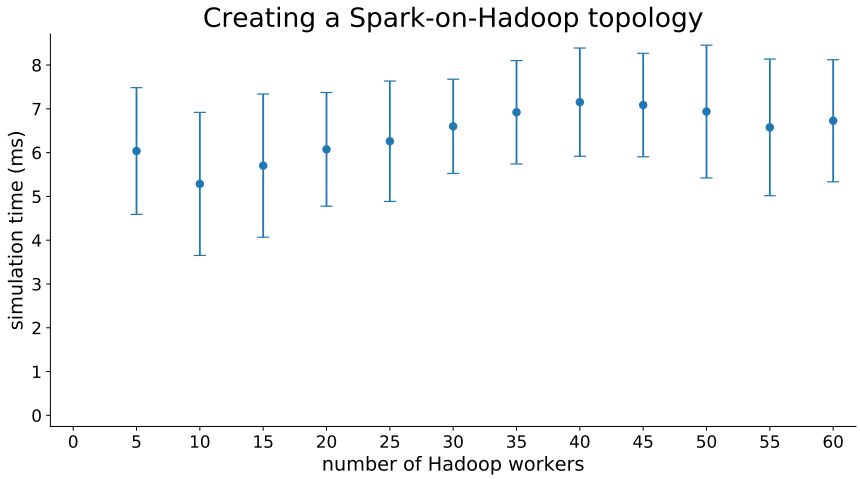

Figure 2.9: The overhead of the orchestrator conversation is less than 0.10 seconds which is insignificant compared to a minimum deployment time of around 20 minutes. The simulation time remains constant because the amount of workers is represented in the Hadoop Worker service agent as an integer value.

2.4.4 Scalability of the orchestrator conversation

This series of evaluations focuses on how the orchestrator conversation scales when creating the topologies for multiple different cloud applications at the same time. It is a common scenario that cloud infrastructure is shared over multiple teams, projects, and environments such as development, staging and production.

These evaluations simulate such a scenario by deploying multiple Spark-on-Hadoop clusters simultaneously. It does not matter what the actual orchestrated clusters are since these evaluations focus on the scalability of the orchestrator conversation and not the scalability of the orchestration actions themselves.

Unlike scaling a single cluster, creating multiple clusters actually creates multiple orchestration agents. Figure 2.10 shows that the simulation time scales linearly in function of the number of clusters. This linearity is expected since the clusters do not have any dependencies on each other. Consequently, if all the clusters are equal, the time to create the topology for cluster $n+1$ is equal to that for cluster n, thus each cluster adds a constant overhead to the overall simulation time.

2.4.5 Concurrency in the orchestrator conversation

Concurrency is a strong point of a distributed service orchestrator because the orchestrator conversation allows all orchestration agents to run at the same time. This is inherent to the orchestrator conversation and does not require code changes in the orchestration agents. This stands in stark contrast with the monolithic nature of current state of the art orchestrators. A single-

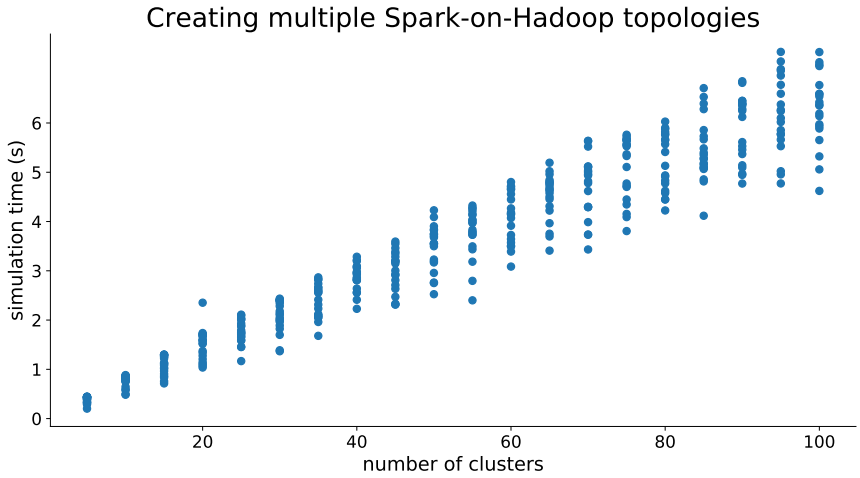

Figure 2.10: The simulation time scales linearly as a function of the number of unconnected clusters of orchestration agents. This graph shows the result of 10 runs for each x value.

process orchestrator cannot orchestrate a topology concurrently, and enabling parallelization in a monolithic orchestrator creates a complex system that is hard to adapt and fine-tune [34]. This series of evaluations has the goal to find out if the orchestrator conversation actually uses the concurrency potential of the topology.

These evaluations use a simple orchestration agent that creates a number of children and waits until these children are ready. This process continues recursively until the requested amount of orchestration agents is created. This creates a tree of orchestration agents connected by the operator relationship. The leaves of the tree immediately go into the ready state as soon as they are created. This causes the ready states to propagate through the tree from the leaves to the root. When the root orchestration agent is in the ready state, the simulation stops and the duration of the simulation is used for evaluation. The duration of two topologies with the same number of nodes but different concurrency potential is compared.

Changing the number of children of an orchestration agent has a big impact on the concurrency potential of the topology. This simulation uses a unary tree of the aforementioned orchestration agents as a topology without concurrency potential. Both the creation of the unary tree and the propagation of the ready state has to happen sequentially, one node after the other. A binary tree is used as an example of a topology that has a lot of potential for concurrency.

The big performance increase when concurrency is possible, as depicted in Figure 2.11, shows that the distributed service orchestrator does indeed use the concurrency potential of the topology. This has a big impact on the real-world performance of a distributed service orchestrator because the main use of the operator relationship is to abstract; to connect one operator to multiple executors, allowing the executors to run concurrently. Note that in a real-world topology, the number

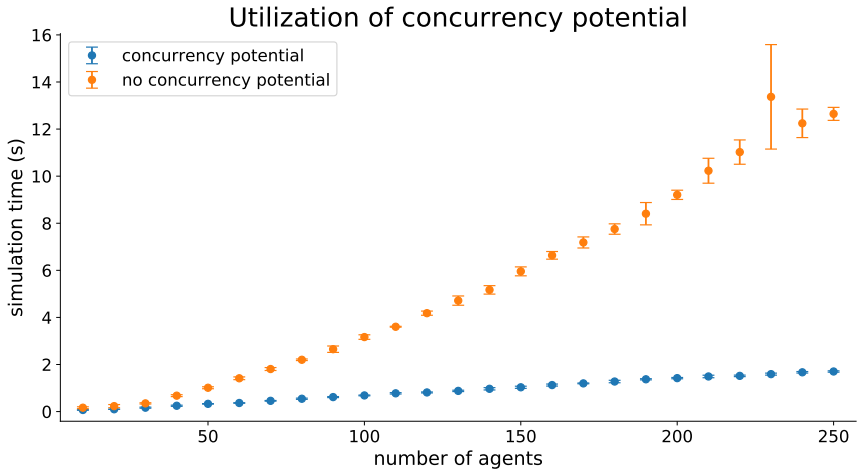

Figure 2.11: The orchestrator conversation happens as concurrently as the topology allows, without any code change required, which significantly improves the scalability. Each dot represents the aggregated results of 5 simulation runs.

of children can differ between orchestration agents and can exceed two, as is the case with the Hadoop OA in the Spark-on-Hadoop cluster, which has 4 children. A topology where each node has 3 or more children contains even more concurrency potential, but our simulations have shown no significant performance improvements between a binary and tertiary tree because at that point, the number of CPU cores is the bottleneck, not the number of concurrent threads.

The optimal usage of concurrency potential also explains the scalability of orchestrating multiple unconnected clusters as discussed in Subsection 2.4.4: the clusters are not dependent on each other, thus the orchestration of all clusters runs concurrently.

2.5 Conclusion and Future Work

This article proposes the orchestrator conversation as a way to introduce abstraction of the cloud application to topology-based cloud modeling languages. The focus on the conversation instead of the orchestrator itself makes it possible for software vendors to create components that translate abstract, declarative models into management actions on their software. The evaluation shows that the orchestrator conversation can be used to create much smarter abstractions than possible with the state of the art, the overhead of the conversation is minimal compared to the actual time to deploy the cloud application, and that the request model is indeed less complex while the underlying full topology is still present. The evaluation also shows that the decentralized nature of the conversation enables the management of the cloud application to happen inherently concurrent. This greatly enhances the scalability of the solution and alleviates the need for the

complex and error-prone process of manually programming concurrency into an orchestrator.

Hierarchical abstraction layers are possible by creating a tree of "operator" relationships. System administrators specify their request in the same way as orchestration agents: using the operator relationship, which means that orchestration agents themselves can fully utilize the abstractions of other orchestration agents in order to reason about and manage the cloud application on a higher level. This also makes it possible for system administrators to gradually automate more and more management tasks by creating orchestration agents that use the operator relationship to drive the behavior of other agents. It is important to note that this approach of a conversation instead of an orchestrator does not take orchestrator vendors out of the picture. Markets can still emerge around creating and selling orchestration agents where orchestrator vendors can use their expertise to create auto-scalers or specialized orchestration agents that can take SLA and QoS requirements into account.

This article treats OAs and SAs as a black box, however it is valuable for future work to see how QoS and SLA requirements can be reasoned about in such a distributed manner. The state machine approach currently used by cloud modeling languages to model the lifecycle of a single component is not flexible enough for agents with multiple independent sub-states. Future work will investigate more flexible ways to model SAs and OAs and investigate the advantages of the orchestrator conversation when used as an alternative rather than a supplement to existing cloud resource orchestration frameworks. An interesting topic to further investigate is whether this decentralized nature has a positive impact on the solution's ability to cope with network segmentation and its resiliency. Another important question is whether the hierarchical nature of the operator relationship has an adverse effect on the solution's ability to self-heal.

Acknowledgments

This research was performed partially within the FWO project "Service-oriented management of a visualized future internet".

Bibliography

[1] DICE-Deployment-Examples: Example blueprints used with the DICE Deployment Service, October 2016. https://github.com/dice-project/DICE-Deployment-Examples. URL: https://github.com/dice-project/DICE-Deployment-Examples.

[2] Vasilios Andrikopoulos, Santiago Gómez Sáez, Frank Leymann, and Johannes Wettinger. Optimal Distribution of Applications in the Cloud. In *Advanced Information Systems Engineering*, pages 75–90. Springer, Cham, June 2014. doi:10.1007/978-3-319-07881-6_6.

[3] Benoy Anthony, Konstantin Boudnik, Cheryl Adams, Branky Shao, Cazen Lee, and Kai Sasaki. Ecosystem at Large: Hadoop with Apache Bigtop. In *Professional Hadoop®*, pages 141–160. John Wiley & Sons, Inc, 2016. URL: http://onlinelibrary.wiley.com/doi/10.1002/9781119281320.ch7/summary, doi:10.1002/9781119281320.ch7.

[4] Atos SE. ALIEN 4 Cloud, 2017. URL: https://alien4cloud.github.io/.

[5] Bigtop developers. Apache Bigtop Github project, September 2017. https://github.com/apache/bigtop. Accessed October 1, 2017. URL: https://github.com/apache/bigtop.

[6] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, October 2009. doi:10.1109/MC.2009.326.

[7] Denise Bombonatti, Miguel Goulão, and Ana Moreira. Synergies and tradeoffs in software reuse – a systematic mapping study. *Software: Practice and Experience*, 47(7):943–957, August 2016. doi:10.1002/spe.2416.

[8] Uwe Breitenbucher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 87–96. IEEE, 2014. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6903461.

[9] Antonio Brogi, Andrea Canciani, Jacopo Soldani, and PengWei Wang. A Petri Net-Based Approach to Model and Analyze the Management of Cloud Applications. In *Transactions on Petri Nets and Other Models of Concurrency XI*, Lecture Notes in Computer Science, pages 28–48. Springer, Berlin, Heidelberg, 2016. URL: https://link.springer.com/chapter/10.1007/978-3-662-53401-4_2, doi:10.1007/978-3-662-53401-4_2.

[10] Antonio Brogi, Paolo Cifariello, and Jacopo Soldani. DrACO: Discovering available cloud offerings. *Computer Science - Research and Development*, 32(3-4):269–279, July 2017. URL: https://link.springer.com/article/10.1007/s00450-016-0332-5, doi:10.1007/s00450-016-0332-5.

[11] Antonio Brogi and Jacopo Soldani. Finding available services in TOSCA-compliant clouds. *Science of Computer Programming*, 115:177–198, January 2016. doi:10.1016/j.scico.2015.09.004.

[12] Mark Burgess and Oslo College. Cfengine: a site configuration engine. In *USENIX Computing systems, Vol*, 1995.

[13] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, January 2016. doi:10.1145/2898442.2898444.

[14] Canonical Ltd. Apache Bigtop and Juju: A Charming Approach to Big Data, 2017. URL: http://bigdata.juju.solutions//2016-06-07-apache-bigtop-and-juju/.

[15] Canonical Ltd. conjure-up: Get started with big software, fast, 2017. URL: https://conjure-up.io/.

[16] Canonical Ltd. Ubuntu Juju: Operate big software at scale on any cloud, 2017. URL: https://jujucharms.com/.

[17] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. Di Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič. DICE: Quality-driven Development of Data-intensive Cloud Applications. In *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*, MiSE '15, pages 78–83, Piscataway, NJ, USA, 2015. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2820489.2820507.

[18] Cloudify. Cloudify Plugins documentation., 2018. URL: https://docs.cloudify.co/4.2.0/plugins/overview/.

[19] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Information and Computation*, 239:100–121, December 2014. doi:10.1016/j.ic.2014.11.002.

[20] Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *Proceedings of the 9'th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017.

[21] Christian Endres, Uwe Breitenbücher, Frank Leymann, and Johannes Wettinger. Anything to Topology – A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts. In *{Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal*, Porto, Portugal, April 2017. SCITEPRESS. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2017-23%20-%20Anything%20To%20Topology%20A%20Method%20and%20System%20Architecture%20to%20Topologize%20Technology-Specific%20Application%20Deployment%20Artifacts.pdf.

[22] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, and Noel De Palma. Reliable self-deployment of distributed cloud applications. *Software: Practice and Experience*, 47(1):3–20, 2017. doi:10.1002/spe.2400.

[23] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing*, 3(5):81–88, September 2016. doi:10.1109/MCC.2016.112.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. doi:10.5555/1972457.1972488.

[25] C. Hounkonnou and E. Fabre. Empowering self-diagnosis with self-modeling. In *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*, pages 364–370, October 2012.

[26] Johannes Kirschnick, Jose M. Alcaraz Calero, Patrick Goldsack, Andrew Farrell, Julio Guijarro, Steve Loughran, Nigel Edwards, and Lawrence Wilcock. Towards an Architecture for Deploying Elastic Services in the Cloud. *Softw. Pract. Exper.*, 42(4):395–408, April 2012. doi:10.1002/spe.1090.

[27] Chris Lauwers. Declarative vs. Imperative Orchestrator Architectures, June 2017. http://blog.ubicity.com/2017/06/declarative-vs-imperative-orchestrator.html. Accessed August 18, 2017. URL: http://blog.ubicity.com/2017/06/declarative-vs-imperative-orchestrator.html.

[28] E. Lavinal, T. Desprats, and Y. Raynaud. A multi-agent self-adaptative management framework. *International Journal of Network Management*, 19(3):217–235, May 2009. URL: http://onlinelibrary.wiley.com/doi/10.1002/nem.699/abstract, doi:10.1002/nem.699.

[29] Miguel Caballer, Marco Tangaro, Marica Antonacci, and Alfonso Pérez. tosca-types: YAML description of new types added to extend TOSCA Simple Profile in YAML Version 1.0, June 2017. URL: https://github.com/indigo-dc/tosca-types.

[30] Derek Palma, Matt Rutkowski, and Thomas Spatzier. TOSCA Simple Profile in YAML Version 1.0, August 2015. OASIS Committee Specification Draft 04 / Public Review Draft 01. URL: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0-csprd01.html.

[31] Pushan Rinnen, John McArthur, Stanley Zaffos, Raj Bala, Dave Russell, Julia Palmer, Valdis Filks, Garth Landers, Santhosh Rao, Rhame Rhame, Julian Tirsu, and Shane Harris. 2017 Strategic Roadmap for Storage. Research Note G00324339, March 2017. Published by ClearSky Data, Inc. URL: https://www.gartner.com/doc/3632117/-strategic-roadmap-storage.

[32] Dave Russel and Mike Chuba. Top Challenges Facing I&O Leaders in 2017 and What to Do About Them. Technical Report G00324370, Gartner, February 2017. URL: https://www.gartner.com/doc/3615217/top-challenges-facing-io-leaders.

[33] D. Salomoni, I. Campos, L. Gaido, G. Donvito, M. Antonacci, P. Fuhrman, J. Marco, A. Lopez-Garcia, P. Orviz, I. Blanquer, M. Caballer, G. Molto, M. Plociennik, M. Owsiak, M. Urbaniak, M. Hardt, A. Ceccanti, B. Wegh, J. Gomes, M. David, C. Aiftimiei, L. Dutka, B. Kryza, T. Szepieniec, S. Fiore, G. Aloisio, R. Barbera, R. Bruno, M. Fargetta, E. Giorgio, S. Reynaud, L. Schwarz, A. Dorigo, T. Bell, and R. Rocha. INDIGO-Datacloud: foundations and architectural description of a Platform as a Service oriented to scientific computing. *arXiv:1603.09536 [cs]*, March 2016. arXiv: 1603.09536. URL: http://arxiv.org/abs/1603.09536, doi:10.48550/arXiv.1603.09536.

[34] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM. doi:10.1145/2465351.2465386.

[35] Merlijn Sebrechts. Orchestration Agents Code on Github, October 2017. URL: https://github.com/IBCNServices/oa.

[36] Merlijn Sebrechts, Sander Borny, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Model-driven deployment and management of workflows on analytics frameworks. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2819–2826, December 2016. doi:10.1109/BigData.2016.7840930.

[37] Merlijn Sebrechts, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration. In *2016 IEEE International Conference on Mobile Services (MS)*, pages 156–159, June 2016. doi:10.1109/MobServ.2016.31.

[38] Daniel Seybold, Jörg Domaschka, Alessandro Rossini, Christopher B. Hauser, Frank Griesinger, and Athanasios Tsitsipas. Experiences of Models@Run-time with EMF and CDO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 46–56, New York, NY, USA, 2016. ACM. doi:10.1145/2997364.2997380.

[39] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A Runtime Model Based Monitoring Approach for Cloud. pages 313–320. IEEE, July 2010. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5557977, doi:10.1109/CLOUD.2010.31.

[40] José Manuel Sánchez Vílchez, Imen Grida Ben Yahia, Chidung Lac, and Noel Crespi. Self-modeling based diagnosis of network services over programmable networks. *International Journal of Network Management*, 27(2):n/a–n/a, March 2017. doi:10.1002/nem.1964.

[41] TOSCA Technical Committee. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee | Charter, December 2013. URL: https://www.oasis-open.org/committees/tosca/charter.php.

[42] K. Tsakalozos, C. Johns, K. Monroe, P. VanderGiessen, A. Mcleod, and A. Rosales. Open big data infrastructures to everyone. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2127–2129, December 2016. doi:10.1109/BigData.2016.7840841.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. doi:10.1145/2741948.2741964.

[44] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432.

[45] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.*, 50(2):26:1–26:41, May 2017. doi:10.1145/3054177.

[46] Johannes Wettinger, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann. Collaborative gathering and continuous delivery of DevOps solutions through repositories. *Computer Science - Research and Development*, 32(3-4):281–290, July 2017. URL: https://link.springer.com/article/10.1007/s00450-016-0338-z, doi:10.1007/s00450-016-0338-z.

3

Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications

This chapter tackles the second research question: "How to encapsulate and reuse system administrator's knowledge about when to perform which management actions?" It proposes the reactive pattern to make it easier to create custom lifecycles to manage a single component of a composed application. The pattern is specifically designed to make knowledge sharing between developers of lifecycles easier: it allows these developers to create reusable layers of lifecycle steps. This chapter ends with an evaluation of the ecosystem that existed two years after the initial release of software implementing this pattern, showing a healthy amount of reuse and collaboration between different creators. Even in 2022, this research is still relevant, for example as inspiration for how to increase code sharing between custom Kubernetes controllers. Even a modern framework to create custom orchestration logic for Kubernetes controllers, such as the operator SDK, lacks tools to encapsulate and share parts of this logic.

⋆ ⋆ ⋆

M. Sebrechts, C. Johns, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck

Published in the proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018

Abstract Automated management and orchestration of cloud applications have become increasingly important, partly due to the large skills shortage in IT operations and the increasing complexity of cloud applications. Cloud modeling languages play an important role in this, both for describing the structure of a cloud application and specifying the management actions around it. The TOSCA cloud model standard recently defined *declarative workflows* as the preferred way to specify these management actions but, as noted in the standard itself, this is far from ideal. This paper draws lessons from six years of using declarative workflows in Juju for deploying and managing complex platforms such as OpenStack and Kubernetes in production. This confirms the limitations: declarative workflows are inflexible, hard to reuse, and allow for related components to become silently incompatible. This paper proposes the *reactive pattern* to solve these issues by enabling the creation of *emergent workflows* using declarative *flags* and *handlers*, which can be easily grouped into reusable layers. After more than two years of using this pattern in production as part of our *charms.reactive* framework, it is clear that it enables reusability and ensures compatibility: 67% of reactive charms share parts of the management workflow and 73% of reactive charms share a relationship workflow.

3.1 Introduction

Due to the large skills shortage in IT operations [9] and the increasing complexity of cloud applications [16], automated management and orchestration of cloud applications have become increasingly important. The OASIS *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [13] is a front-runner in this field: a standard with large backing from both the industry and academia. It provides a specification to create self-contained cloud models that describe the structure of a cloud application in a topology model, as well as the surrounding management and orchestration processes in a workflow model. The order in which these processes are to be executed is either explicitly defined in an imperative workflow model or implicitly in a declarative workflow model. The latter type is of great importance to this research because it allows capturing the knowledge on how to manage a cloud application in a reusable way, which is crucial to solving the skills shortage and the complexity of cloud applications [16]. Consequently, the Juju cloud modeling language [4], which closely resembles the TOSCA standard, has been using declarative workflows since its inception. Section 3.2 introduces these concepts and related work in greater detail.

However, as noted in the TOSCA specification itself, declarative workflows are inherently inflexible: every workflow needs to adhere to a single lifecycle defined by the cloud modeling language.

The real-world implications of this issue have become painfully clear during 6 years of managing complex platforms in production with Juju. Section 3.3 reflects on this experience and identifies the main shortcomings of the declarative approach.

This paper proposes the reactive pattern in Section 3.4 to address the limitations of declarative workflows. Specifically, the reactive pattern allows creating custom-fit workflows, not limited by lifecycles, and enables more fine-grained reuse than declarative workflows. Its implementation is discussed in Section 3.5 and the evaluation of two years of production use is discussed in Section 3.6. The reactive pattern has resulted in widespread code reuse and increased compatibility. Consequently, it forms a great battle-tested foundation for improved workflow support in the TOSCA standard.

3.2 Background and Related Work

3.2.1 Cloud Modeling Languages

Model-based management of cloud applications is ubiquitous [16] and can be traced back to the 1995 paper of Burgess et al. where the idea of converging towards a predefined end-state was proposed [1]: a system administrator declaratively specifies what the desired end-state of the cloud application is, and an orchestrator interprets that specification and iteratively executes the required actions to get the application into that state. This idea has evolved over the years and has resulted in the creation of topology-based cloud modeling languages to enable better portability and reusability [13]: the model of the cloud application consists of a graph of components which are connected by their dependencies. Each component is a self-contained description of part of the cloud application and new models can be created by rearranging the components thus making each component reusable.

As said in the introduction, OASIS TOSCA [13] provides a specification to create self-contained cloud models that describe both a) the structure of a cloud application in a topology model, and b) the management and orchestration processes surrounding it in a workflow model. The topology consists of a number of **nodes** connected to each other using **relationships**. Relationships denote dependencies between two nodes. A web app, for example, might have a relationship with a database to denote that the web app uses the database for storage. The types of relationships possible between nodes are defined by their **node type** in the form of requirements and capabilities: a relationship connects a node that requires a certain dependency to a node that provides the same dependency. The structure of the workflow model has changed over the past few years. TOSCA 1.0, released in 2013, does not enforce a specific workflow language but favors BPMN. TOSCA Simple YAML profile 1.0, released in 2016, lost the ability to specify a workflow and only with the 2018 release of TOSCA Simple YAML profile 1.1 have workflows been included again, now in two forms: imperative and declarative workflows [10].

3.2.2 Imperative Workflows

Imperative workflows are often depicted as a set of *activities* linked by a *control flow*. Each activity is a piece of work that forms one logical step of a process. The control flow describes the order in which the individual activities are performed. These can be represented as a directed graph where each node represents an activity and the vertices describe the control flow. In an imperative workflow, the order of execution is explicitly defined as part of the workflow definition e.g. as a flow diagram. Many IT service management practices such as Information Technology Infrastructure Library (ITIL) [2] use imperative workflows as high-level descriptions of IT business processes. Thus, the use of such workflows in cloud modeling languages makes it easy to align IT services with business needs. The imperative workflows in a cloud model define how to deploy, manage and undeploy a topology. Each activity is a management action such as "install MySQL", and the control flow describes when each management action needs to be performed. During deployment, the orchestrator executes the workflow step by step until the entire topology is deployed.

The downside of using imperative workflows is that they are defined for a specific topology instead of for one component. Thus, when the topology is changed, the workflow needs to be recreated. This is an inherent limitation of imperative workflows: changing a constraint in an imperative workflow description requires a complete rewrite of the control flow [6]. Having to rewrite the workflow every time a component in the topology changes, goes against the modular nature of topology-based cloud modeling languages. Wagner et al. propose to define the imperative workflows on the level of the individual nodes, and interconnect the workflows of all the components in a topology using a choreography. This approach however still requires manual creation of the choreography because the orchestrator cannot know how the individual workflows should be connected [15].

3.2.3 Declarative Workflows

Declarative workflows provide optimal reusability: the declarative workflow for each component of a topology is contained inside the description of that node. Adding the node to the topology will automatically add all the management activities to the global workflow. This is because the control flow is not explicitly specified but rather implicitly derived from the constraints of each activity. In TOSCA, the constraints specify which lifecycle phase the activity is part of, for example *installing*, *configuring* or *starting*. It is then up to the orchestrator to decide when each lifecycle phase for each component needs to be executed, so the orchestrator "generates" an imperative workflow by merging all activities from all nodes in the topology [3]. The lifecycles themselves are however defined by the orchestrator which presents the biggest drawback of declarative workflows in TOSCA: workflows are limited to the states and transitions defined in the orchestrator's lifecycle.

Furthermore, this also restricts the types of dependencies possible between nodes. TOSCA specifies a number of normative relationships that each carry specific meaning about the dependency

between related nodes. As an example, the *DependsOn* relationship means that the target node needs to be started before the source node is created. This directly translates into how the orchestrator connects the declarative workflows of these two components: the deployment workflow of the source node is executed when the target node reaches the *started* state. As a result, declarative workflows can only model dependencies which are explicitly defined by the orchestrator. The current TOSCA specification, for example, does not support circular dependencies [5], i.e. dependencies where the control flow jumps back and forth between two nodes multiple times.

3.3 Lessons learned: history of declarative workflows in Juju

Juju [4] is a cloud modeling language and orchestrator created by Canonical that closely resembles the TOSCA standard. Since its inception in 2012, Juju has been used in production to deploy big software such as OpenStack and Big Data clusters [14], and is at the core of BootStack and the Canonical Distribution of Kubernetes.

A Juju **charm** is similar to a TOSCA node type: it represents one service in the cloud application and defines which relationships it supports using **requires** and **provides** statements. Juju also uses declarative workflows: the orchestrator defines a number of lifecycle stages such as *install*, *start* and *config-changed*, and executes a program called a **hook** during each lifecycle stage. A hook is a workflow activity and its name defines which lifecycle transition it performs. Thus, the Juju orchestrator decides *when* hook code gets executed, and the charm developer decides *what* operation should be performed. A deployed instance of a charm is called a **unit** and adding a unit to a model automatically adds the hooks of its charm to the topology-wide declarative management workflow. This approach resulted in a number of issues.

The lifecycle provided by Juju does not match the actual lifecycle of the managed services. Juju's provided lifecycle is too simple for most services, which require many more lifecycle phases and transitions. This results in a frequently used anti-pattern where all lifecycle stages execute the exact same code which implements a rudimentary state machine with *if-then* statements that mimics the real lifecycle of the application. The state machine figures out which actual lifecycle stage the application is in, and executes the required actions. Expanding the lifecycle of Juju's orchestrator is not a good solution because each service requires its own specialized lifecycle so a one-size-fits-all lifecycle is simply not sufficient. Moreover, it should not be up to the orchestrator to define what the lifecycle of a service is, this should be defined by the service.

Reusing parts of the lifecycle of a single service is difficult. Many services share components, and many lifecycle steps are the same for multiple services. Many services are installed using the distribution package manager, for example, and need to be updated when security fixes are released. Encapsulating this functionality in a way that it allows being reused in other lifecycles is not possible. Over the years, a number of charm helper libraries have been created in order to increase code reuse, but the issue with a library is that it only encapsulates *how* to do a certain lifecycle action, not *when* that action should be performed.

The relationship lifecycle provided by Juju does not match the actual relationship lifecycle of the managed services. As explained in Section 3.2, the use of declarative lifecycles restricts the types of dependencies between two nodes to the ones supported by the orchestrator. Juju supports only one type of dependency in which the lifecycles of both units run concurrently. After the *start* hook of both units, the relationship lifecycle runs and the units exchange configuration values. In reality, however, many services require knowing configuration values, such as the IP address of a database, before starting. This causes developers to create a state machine that completely ignores hooks such as *config-changed* and *start*, and waits until the *relation-changed* hook to actually configure and start the service. This results in a discrepancy between the state that the orchestrator thinks a service is in, and the actual state a service is in.

Silent incompatible relationships. Because of the previous issue, the relationship lifecycles are actually implemented by the charm, instead of by the orchestrator. The orchestrator has therefore no way of verifying that two ends of a relationship actually implement a compatible lifecycle. This has resulted in many semi-compatible charms that implement the same relationship according to the orchestrator, but differ in subtle incompatible ways in practice.

3.4 The reactive pattern

This paper proposes the reactive pattern as a fundamentally new approach to managing services using cloud modeling languages. Such pattern allows the creation of flexible and reusable emergent workflows that manage the entire lifecycle of a modeled cloud application including dependency management, initial deployment, second day operations, topology changes and node type upgrades. Although it was initially created for the Juju cloud modeling language, the pattern itself is generic enough so that it can be used in different cloud modeling languages such as TOSCA or as the *service engine* in a Distributed Service Orchestrator [12]. This section gradually introduces all the primitives of the reactive pattern and explains their role and how they address the shortcomings of declarative workflows.

Just like with regular declarative workflows, the actual management operations are encapsulated in activities which are part of the node definitions. The novel part of this pattern is how the control flow gets created: the orchestrator does not define a lifecycle, it only defines a number of events. Developers create custom event-based workflows for each service and hook them into these events. These workflows are created using constraint-based modeling [8]: each activity defines a set of constraints which need to be satisfied in order for them to execute. Unlike approaches like DECLARE [7], these constraints are not explicitly tied to events regarding the execution of other activities. Rather, the constraints use semantic flags that can also represent a number of different types of events such as the arrival in a certain state, a change in the topology and service events e.g. a crash.

3.4.1 Handlers and Flags

The reactive framework is based on the idea of handlers reacting to flags. Handlers are the activities of the workflow: pieces of code that perform management actions on the cloud service. The control flow, the order in which handlers get executed, is driven by flags: each handler defines which flags it reacts to, i.e. which flags need to be set and/or unset for the handler to execute. The framework executes a handler when its preconditions are met, during which it modifies the service, and can set and clear flags. This triggers other handlers to run, until there are no more handlers whose preconditions are met. In this sense, the reactive pattern uses constraint-based modeling with arbitrary events.

The power of a flag is that it can represent almost anything, from internal state such as "the service is running" and "disk utilization is critically high" to topology modifications such as "a new relation is established" or "this node has been removed". The following is a non-exhaustive list of what semantic meaning a flag might hold.

- **Lifecycle Stage:** The orchestrator itself defines a number of flags that represent which lifecycle transition it requests such as *install*, *config-changed*, and *stop*. These are the reactive pattern's counterpart to the hooks and lifecycles of declarative workflows.
- **Service state:** Developers can define a number of flags that represent low-level state of the service such as *"the webserver is installed"* and *"the SSL certificate is registered"*.
- **Service events:** A flag can also represent events that happened in the past, and that might need to be handled, such as *"the service has crashed"*, which might require notifying a system administrator, even when the service has successfully been restarted.
- **Topology state and events:** Flags can also represent the topology or changes to it. A flag can indicate that a new relation was created in the model or that a related service in the topology has entered a certain state.
- **Day 2 operations:** A flag can signal that a backup is requested, that an update is required, or that an SSL certificate needs to be renewed.

Note that not all flags need to be set by the handlers themselves. The operating system itself can set a flag when a service crashes or when a certain time has passed, and the orchestrator sets flags to indicate which lifecycle stage the application is in and what the current state of the topology is. This for example allows the workflow to hook into the lifecycle provided by the orchestrator.

Definition 3.1. *A **handler** is an activity that manages a cloud resource, accompanied by a set of preconditions that, using flags, states when that activity should execute.*

Definition 3.2. *A **flag** is a boolean identified by a unique string that is a semantic representation of an event to be used in a handler's preconditions.*

Figure 3.1 shows a custom workflow that emerges from a set of handlers and their preconditions. Each activity is a handler and the control flow emerges from their preconditions. The orchestra-

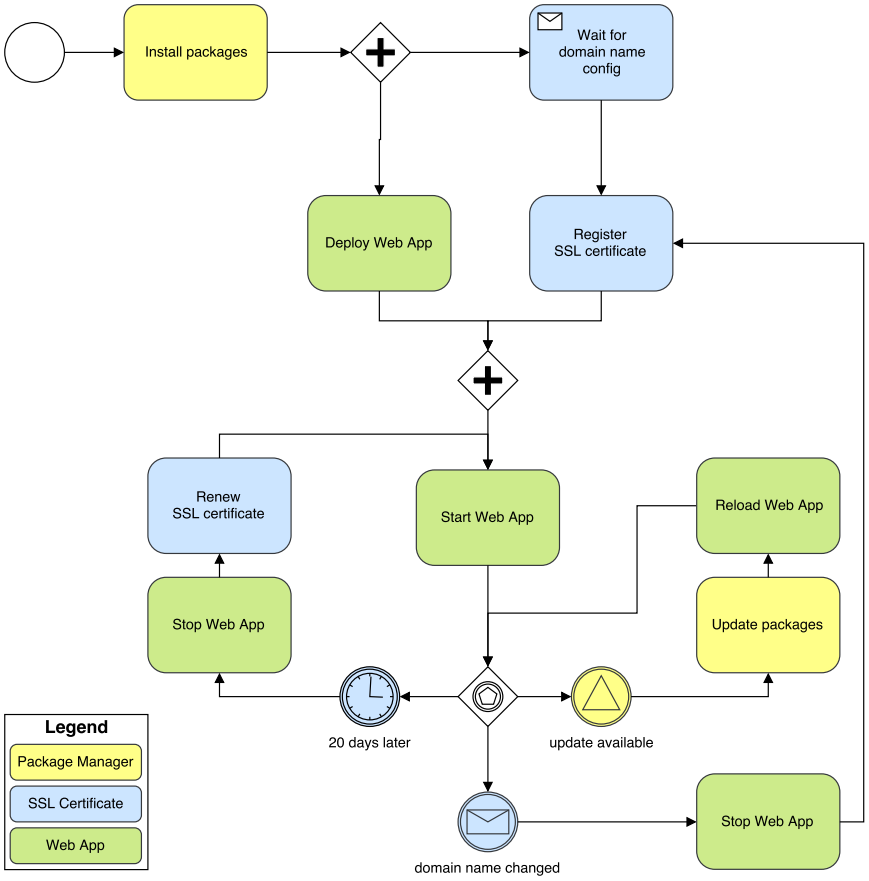

Figure 3.1: This workflow emerges from the handlers, their preconditions and their flags. Each handler is represented by an activity. The "stop web app" handler is represented by two different activities because the next activity depends on which activity was executed previously to "stop web app". The handlers are colored according to which aspect of the service they manage.

tor starts the workflow and sets the appropriate flags when the domain name config is set and changed. The operating system itself sets flags when 20 days have passed and when an update is available to the packages. The pseudocode for the handlers and their preconditions is available online [11].

The resulting workflow shows that some activities such as deploying the web app and registering the SSL certificate can be executed in parallel. This however only regards the control flow dependencies, not the actual dependencies of the activities themselves: it does not matter which action is run first, but the actions might not be able to run at the same time.

In summary, flags and handlers allow the construction of emergent workflows that hook into and

expand the lifecycle provided by the orchestrator. Because the emergent reactive workflow hooks into the lifecycle provided by the orchestrator using flags, it can leverage the existing techniques to combine the declarative workflows of multiple components into a single workflow that manages the entire cloud application. Just like with declarative workflows, reactive emergent workflows are shipped as part of the node type of a component. When that component gets added to a model, the accompanied workflow will be hooked into the model's global workflow. This approach thus eliminates the downsides of TOSCA's imperative workflows while allowing for a greater level of flexibility.

3.4.2 Scope

A big advantage of cloud modeling languages stems from the separated scope between nodes. A node can only access information about another component if there is an explicit relationship that shares that piece of information. This property is also present in the reactive pattern. Flags in the reactive pattern are unit-scoped: each instance of a node type has its own set of flags. Handlers themselves are node-type scoped: all instances of a single node-type have the same set of handlers. This means that the emergent workflow of each unit will be the same, but the current position in the workflow might be different. The web app example from Figure 3.1 is a single service that consists of a number of components: an SSL encrypt certificate, a webserver and a web app. When the web app scales out into multiple instances, each instance will have its own set of flags, but the handlers will be the same over each unit.

3.4.3 Layers

As mentioned previously, the web app example service can be divided into three components: the webserver, the SSL certificate and the web app. The emergent workflow in Figure 3.1 shows each activity colored based on which component it manages. As one can see, it is not possible to divide the emergent workflow into three sub-workflows, one for every component. With the reactive pattern, this becomes possible since the workflow itself is just an emergent property from the handlers and their preconditions: it only exists at runtime. At design time, the handlers can be divided into arbitrary groups because there are no explicit dependencies between activities: the only dependencies are implicit with the flags as an intermediary.

In the reactive pattern, each set of grouped handlers is called a "layer". Figure 3.2 shows the handlers from the web app example divided into three layers. Each layer contains the handlers that manage a specific part of the service: the web app, the SSL certificate and the webserver. Adding a layer to a node type results in the handlers of that layer being added to the emergent workflow of that node type. This thus greatly improves the reusability and allows developers to focus on the components that they are an expert in, instead of having to code the entire service. In a sense, this is aspect-oriented programming: each layer contains the activities that manage one specific aspect of the service. The flags define the "cut points", the points in which the aspects get injected into the program code.

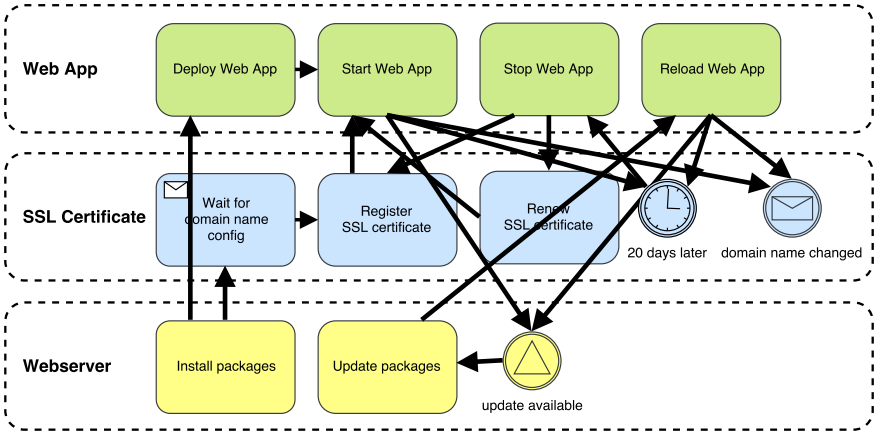

Figure 3.2: Since the dependencies between handlers are implicit, the handlers can be grouped by which aspect of the service they manage, even though the emergent control flow goes back and forth between the layers in an erratic manner.

A layer is one level below a TOSCA node type: multiple layers combined form one node type. From the viewpoint of the orchestrator, all layers of the same node type share the same lifecycle. The orchestrator does not coordinate the lifecycles of each layer individually since the control flow of a service is defined by the flags and the preconditions of handlers. This also has the advantage that a model designer does not come into contact with layers, the model designer only sees a single node and layers are an "implementation detail" of the node. Finally, this makes it possible to use layers without needing any changes to TOSCA itself since layers are "compiled" into a TOSCA node type, and the orchestrator only interacts with the node type.

In order for layers to be reusable, it is important that each layer defines what the semantic properties are of each flag. Some flags might be for internal use in a layer itself, while other flags are to be used by other layers to signal this layer or to use in the preconditions of their handlers. It is also important to define how these flags will be managed by a layer: whether or not flags will be automatically set or removed when certain conditions happen.

Furthermore, it is important to avoid conflicts between layers. An example of a conflict is when two handlers, A and B, react to the same flag, *config.changed*, and both clear that flag during execution. Though it is not immediately obvious, this results in non-deterministic behavior because the workflow is executed sequentially and the preconditions of flags are rechecked after execution of every handler. If handler *A* runs first, it will clear the *config.changed* flag and handler *B* will not even run. This in itself is wanted behavior: a handler is never allowed to run if its preconditions are not met. If handler *A* clears a flag during its execution, it signals that the event that set the flag is *handled*, indicating that no other handlers which handle the same event should run.

It is however entirely possible that multiple layers handle the same event. Layers do not have explicit dependencies on each other, so a handler cannot know, at the time of clearing the flag, if

the event is actually handled by every layer. Triggers are used to avoid such conflicts.

Definition 3.3. *A **trigger** is a causal, directed dependency between two flags that sets or clears a flag immediately when the other flag is set or cleared.*

Immediately in this context means that when a flag changes, the execution of handlers is paused until all triggers are processed.

Using a trigger, a layer links the *config.changed* flag to a custom flag for example *layer-a.config.changed*, such that the custom flag is set immediately after *config.changed* is set. This custom flag is meant for internal use in that layer only and is thus prefixed with the layer's name. Since a trigger is one-way, the custom flag will *not* be cleared when another layer clears *config.changed*. The developer can thus safely use the custom flag in the preconditions of a handler without having to worry that it will be cleared by another layer before the handler has a chance to run.

3.4.4 Interface layers and Endpoints

Much like layers contain reusable handlers to manage individual services, interface layers contain reusable handlers to manage the relationship *between two nodes*. Unlike regular layers, a single interface layer contains handlers for two nodes because an interface layer implements both sides of the relationship. An interface layer is thus a declarative model of the communication between two nodes. This again makes it possible for the orchestrator to know whether a relationship between two nodes is possible: if two nodes share the same interface layer, the relationship is possible.

Relationships in TOSCA serve two purposes during orchestration: they are used to connect the control flow of two nodes in a way that the dependency is resolved, and they are used to exchange information such as IP addresses in order to configure both services correctly.

The control flow of a single component in the reactive pattern is defined by the flags. It is however not desirable to share all the flags of one node with another node, since that creates deep dependencies between nodes, loses the modularity and limits the reusability of a layer. Thus, all sharing of state and data happens explicitly by the handlers of the interface layer so that the dependencies between the implementation of two nodes are limited to the handlers of the interface layer, which is not an issue since the interface layer is already shared between two nodes.

Each time the relationship and its data gets changed, the orchestrator notifies the interface layer by setting flags that denote lower-level relationship events such as **endpoint.x.joined**, when a relationship is established, **endpoint.x.changed**, when relationship data changes, and **endpoint.x.departed** when a relationship is removed. These lower-level flags are the internal API of an interface, they are only to be used by relationship handlers which react to these events, read and write relationship data and manage higher-level flags. Regular layers should only react to the higher-level flags since those are regarded as the "external API" of the interface.

As an example, in the MySQL case of an interface which is used to connect a node that provides

a MySQL database to a client, the MySQL side will have the higher-level flags **table.requested**, to denote that a client has requested the creation of a table, and **user.requested** which is set when a client requests the creation of a user account. The layer that manages the MySQL database will contain a number of handlers that react to these flags to create the requested tables and users, and will call back to the endpoint object to notify that the requests are executed.

Endpoints are the key to the second purpose for relationships: sharing information between nodes. An endpoint is an object that represents one side of the relationship. It publishes and reads the relationship data to communicate with the endpoint at the other side and it translates the raw relationship data into high-level objects to be used by handlers.

3.5 Implementation

3.5.1 The "charms.reactive" framework

The *charms.reactive* framework is our implementation of the reactive pattern built on top of Juju's declarative workflows[1]. It is written in Python 3. Handlers are decorated python functions or executable files that implement the *external handler API*.

A reactive charm is built from layers. Each layer is a directory with a number of handlers and a *layer.yaml* file that holds metadata such as the name of the layer, and the dependencies of this layer i.e. what other layers this layer uses. The *charm build* tool is used to *compile* a layer and its dependencies into a charm. It downloads all the dependencies from the *layer-index*[2], merges all the layers and packages the result into a deployable charm.

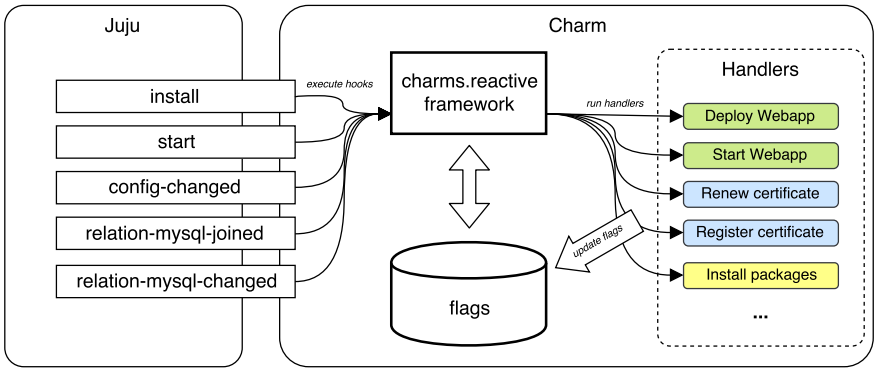

Figure 3.3: The architecture of the charms.reactive framework: when the orchestrator executes a hook, the reactive framework initiates and runs the handlers whose preconditions are true.

Figure 3.3 shows the architecture of the framework. Much like the state machines mentioned in

[1] https://charmsreactive.readthedocs.io/en/latest/
[2] https://github.com/juju/layer-index

Section 3.2, the framework ties into the Juju hooks so that any hook simply executes the framework. It then decides which handlers to run based on the preconditions of the handlers. When there are no more handlers to run, the framework exits the hook. According to the Juju orchestrator, a reactive charm is thus no different from a regular charm.

At the start of each hook, the reactive framework loads the flags from persistent storage, sets and clears the managed flags based on the hook and the information from the orchestrator, and starts to execute the handlers whose preconditions are met, as shown in Listing 3.1. A handler is considered *matching* if the preconditions are true and the handler did not yet run in the current hook or the flags referenced in its preconditions have changed since the last time it ran.

Listing 3.1: Pseudocode for a run of the reactive framework

```
set and clear managed flags
add matching handler to the queue

while queue is not empty:
    for each handler in queue:
        run handler
        if handler failed:
            revert flag changes
            fail hook
        remove handler from queue
        remove not matching handlers from queue
    add matching handler to queue
    if max iterations reached:
        revert flag changes
        fail hook
```

All handlers on the same unit are executed sequentially, even if the emergent control flow allows concurrent execution, since the reactive pattern does not provide a way for handlers to define whether or not two handlers can actually run concurrently. This is however still an improvement over TOSCA's declarative workflows, where even the activities of related nodes are run sequentially. The order in which the reactive framework runs handlers when multiple handlers match is undefined but deterministic: every run will result in the same order, but a charm developer should not rely on any order.

From Juju's standpoint, a hook is transactional: if a hook fails, Juju will rollback the state changes of that hook and try the hook again. This fixes transient failures. For this reason, the reactive framework itself also rolls back all changes to flags when a handler fails. This protects against transient failures as shown by Wettinger et al. [17]. Juju's approach to this does not eliminate the need for idempotency because the orchestrator does not roll back the actual changes to the service so the service might be in an inconsistent state, and handlers might run multiple times when the hook is retried.

3.5.2 Lessons learned

In the initial version of the *charms.reactive* framework, **flags were called *states***, which confused developers because they thought they were building finite state machines (FSMs). It is possible to build an FSM with the reactive pattern, but the pattern is a lot more powerful since it also allows event-based programming. Flags are a much more neutral term which does not imply any specific model of computation. As an example, some of the relationship flags represent events instead of states. The *relationship.{name}.joined* flag is a state: it is set when the relationship reaches the *joined* state, and is cleared when the relationship leaves that state. However, the *relationship.{name}.departed* flag represents an event: it is set every time a unit departs from a relationship and is manually cleared by a handler that "handled" the departure. In contrast, if this event were a state, it would be set when the first unit departs a relationship and never be cleared since that unit remains departed, even when that departure has been "handled".

In the current implementation of the reactive framework, handlers whose preconditions are true are re-executed in every hook. This however turned out to be counter-intuitive for developers, especially new developers without experience writing non-reactive charms. Since the reactive framework is a layer on top of Juju's declarative workflow, and hooks are thus hidden, having such a reliance on their lifecycle adds unnecessary complexity for developers. It is however not possible to change this behavior currently because this will break backwards compatibility.

3.6 In practice

This section shows the results of using the reactive pattern in Juju for more than two years, since our implementation has become available. The data shown in this section is obtained using the public Juju charm store api [3]. A cached copy of the data and the full code to download and process it is available on Github [11].

The charm store contains a total of 529 active charms: charms that have been downloaded in the last month. Of those, only 176 or 33% use the reactive framework. The relatively young age of the framework plays a big role in this: many charms were built before the reactive framework, and porting these charms to the reactive framework is not trivial, since it requires a complete rewrite of the charm code.

Figure 3.4 shows the number of *reused layers and interfaces* per charm, i.e. the number of layers and interfaces which are also used by another charm. This shows the reactive framework has indeed made it possible to reuse workflow code across charms: two-thirds of actively-used reactive charms share at least one layer with another charm. This is an incredibly high number compared to the workflows in TOSCA, where node templates simply can not share any workflow code. However, there is a lot of unused potential because 41% of layers are used in only one charm as shown in Figure 3.5.

[3] https://github.com/juju/charmstore/blob/v5-unstable/docs/API.md

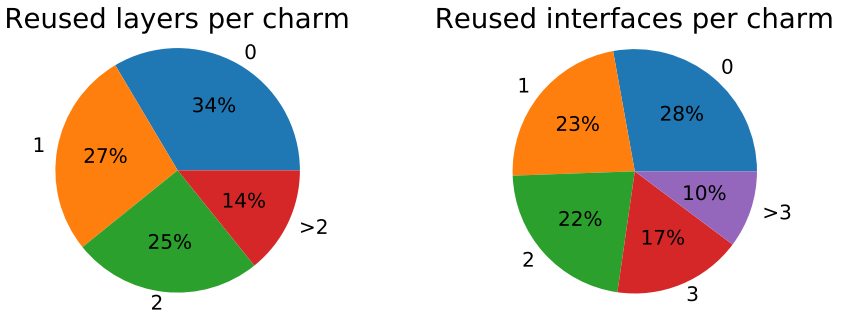

Figure 3.4: Number of reused layers and interfaces per charm.

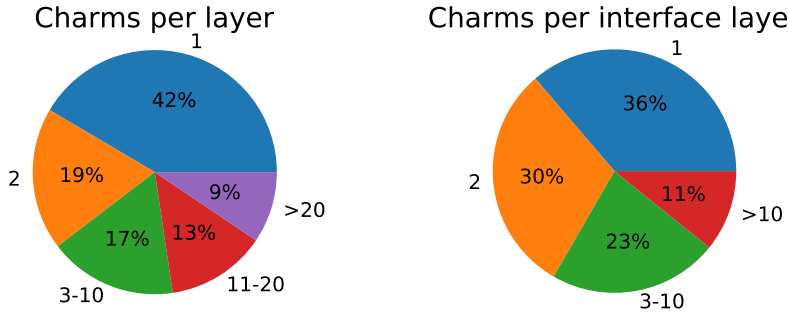

Figure 3.5: Number of times each layer is used.

Figure 3.4 also shows that interface layers are also heavily reused: 73% of charms use at least one interface layer that is shared with another charm, which improves the compatibility of charms implementing the same interface. However, not all interface layers have this benefit: 36% of interface layers are only used once, as shown in Figure 3.5. This is because of the high number of non-reactive charms: these interface layers are used to connect reactive charms to non-reactive charms.

3.7 Conclusion

Six years of managing cloud applications in production using declarative workflows shows that their inflexibility limits their usefulness. Moreover, they do not provide enough opportunity for code reuse, causing duplicated effort and allowing connected workflows managing different services to become silently incompatible. The reactive pattern proposed in this paper addresses these issues by allowing declarative specification of workflows that match the actual lifecycles of the services and by enabling aspect-based grouping of workflow activities into reusable layers.

The results of two years of production use show the reactive pattern's benefits: 67% of reactive

charms use shared layers and 73% of reactive charms use shared interfaces. This shows that the reactive pattern solves the issues of declarative workflows and even though it originated from the Juju ecosystem, it is generic enough so that it can form the basis for improved workflow support in other cloud modeling languages such as TOSCA.

Acknowledgment

This work was supported by the Research Foundation Flanders (FWO) under Grant n° G059615N - "Service-oriented management of a virtualised future internet".

Special thanks to Alex Kavanagh and Stuart Bishop for many insightful discussions on the reactive pattern and their contributions to the charms.reactive framework.

Bibliography

[1] Mark Burgess and Oslo College. Cfengine: a site configuration engine. In *USENIX Computing systems, Vol*, 1995.

[2] Cabinet Office. *ITIL Service Strategy 2011 Edition*. The Stationery Office, Norwich, 2011.

[3] Domenico Calcaterra, Vincenzo Cartelli, Giuseppe Di Modica, and Orazio Tomarchio. Combining TOSCA and BPMN to Enable Automated Cloud Service Provisioning. pages 187–196, February 2018.

[4] Canonical Ltd. Ubuntu Juju: Operate big software at scale on any cloud, 2017. URL: https://jujucharms.com/.

[5] Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic deployment of component-based applications. July 2015. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167642315001409, doi:10.1016/j.scico.2015.07.006.

[6] Raghava Rao Mukkamala. *A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs*. PhD Thesis, IT-Universitetet i København, Denmark, 2012. URL: http://www.itu.dk/people/rao/phd-thesis/DCRGraphs-rao-PhD-thesis.pdf.

[7] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, October 2007. doi:10.1109/EDOC.2007.14.

[8] Maja Pesic, MH Schonenberg, Natalia Sidorova, and Wil MP van der Aalst. Constraint-based workflow models: Change made easy. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2007. doi:10.1007/978-3-540-76848-7_7.

[9] Dave Russel and Mike Chuba. Top Challenges Facing I&O Leaders in 2017 and What to Do About Them. Technical Report G00324370, Gartner, February 2017. URL: https://www.gartner.com/doc/3615217/top-challenges-facing-io-leaders.

[10] Matt Rutkowski and Luc Boutier. TOSCA Simple Profile in YAML Version 1.1, January 2018.

[11] Merlijn Sebrechts. reactive-pattern-results: Code and results of 'Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications', May 2018. URL: https://github.com/IBCNServices/reactive-pattern-results.

[12] Merlijn Sebrechts, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration. In *2016 IEEE International Conference on Mobile Services (MS)*, pages 156–159, June 2016. doi:10.1109/MobServ.2016.31.

[13] TOSCA Technical Committee. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Technical Committee | Charter, December 2013. URL: https://www.oasis-open.org/committees/tosca/charter.php.

[14] K. Tsakalozos, C. Johns, K. Monroe, P. VanderGiessen, A. Mcleod, and A. Rosales. Open big data infrastructures to everyone. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2127–2129, December 2016. doi:10.1109/BigData.2016.7840841.

[15] Sebastian Wagner, Uwe Breitenbücher, Oliver Kopp, Andreas Weiß, and Frank Leymann. Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans. In *Cloud Computing and Services Science*. Springer, Cham, April 2016. doi:10.1007/978-3-319-62594-2_12.

[16] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. A Taxonomy and Survey of Cloud Resource Orchestration Techniques. *ACM Comput. Surv.*, 50(2):26:1–26:41, May 2017. doi:10.1145/3054177.

[17] Johannes Wettinger, Uwe Breitenbücher, and Frank Leymann. Compensation-Based vs. Convergent Deployment Automation for Services Operated in the Cloud. In Xavier Franch, Aditya K. Ghose, Grace A. Lewis, and Sami Bhiri, editors, *Service-Oriented Computing*. Springer Berlin Heidelberg, November 2014. doi:10.1007/978-3-662-45391-9_23.

4

Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes

This chapter addresses RQ 3: "How to encapsulate and reuse system administrator's knowledge about composing microservice applications and (re-)configuring their internal dependencies, in a way that fully integrates into a cloud-native ecosystem" It proposes "orcon", which extends the Kubernetes API so users can model dependencies between microservices. It uses these modeled dependencies to automatically reconfigure microservices to adapt to changes in their dependencies. As the name implies, orcon is a successor to the orchestrator conversation proposed in Chapter 2. Specifically, it adapts the orchestrator conversation to the realities of microservice applications running on container orchestrators. Having a separate management agent for each individual component creates a troublesome overhead in a microservice application because of the sheer number and small size of components. As such, the evaluation at the end of this chapter shows how orcon is able to manage relationships with an overhead that is negligible compared to the agent-based approach of Juju.

⋆ ⋆ ⋆

M. Sebrechts, S. Borny, T. Wauters, B. Volckaert, and F. De Turck

Published in IEEE Access, September 2021.

Abstract Smart cities aim to make urban life more enjoyable and sustainable but their highly heterogeneous and distributed context creates unique operational challenges. In such an environment, multiple companies work together with government on applications and data streams spanning several management domains. Deploying these applications, each of which consists of several connected services, and maintaining an overview of application topologies remains difficult. Even though cloud modelling languages have been proposed to solve similar issues, they are not well fit for such a heterogeneous environment because they often require an "all or nothing" approach. Moreover, cloud modelling languages add an additional abstraction layer that rarely supports all features of the underlying platform and make it harder to reuse existing knowledge and tools. This research defines service relationships as the key element to modelling applications as topologies of services. We use this definition to pinpoint what is lacking in the state of the art Kubernetes orchestration tools and provide a blueprint for how relationship support can be added to any orchestrator. We present "orcon", a proof of concept orchestrator that extends the Kubernetes API to allow managing relationships between services by adding metadata to service definitions. Our evaluation shows this orchestrator enables lifecycle synchronization and configuration change propagation with an overhead of only 0.44 seconds per service.

4.1 Introduction

Smart cities have the potential to make urban life more enjoyable and sustainable by introducing deep integration with Internet of Things (IoT) technology. The inherent heterogeneous and collaborative nature of cities creates unique challenges for integrating IoT. Problems cannot be solved in a vacuum: they often require collaboration between multiple competing industry partners, governments, research institutions and the public. A single end-to-end application can have data streams crossing multiple management domains and environments: it can contain components managed by completely different teams on different networks, clouds and data centers. This unique environment acts as a multiplier to operational complexity, making it hard for developers to focus on the applications themselves.

One such smart city project is "City of Things" [34], transforming the city of Antwerp, Belgium, to tackle a wide variety of challenges such as prediction and detection of flooding, improving traffic flows, creating a smart grid and fine-grained monitoring of pollution. Since inter-disciplinary research, citizen science and industry collaboration are key here. The speed of innovation and the exploratory nature of this project only exacerbates the operational challenges, making it hard to get a clear picture of application topologies and how services are connected. Furthermore, due to this nature, individual services are changed often. Adapting related services to these changes often requires manual effort and close collaboration between teams. This speeds down the rate

of innovation.

Cloud modelling languages such as the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) are a great way to model the full topology of an application. Cloud orchestrators can then automatically manage individual services and their relations to each other. However, these languages often follow an all-or-nothing approach in that the entire application needs to be modeled in that language. Moreover, they create an additional abstraction on top of the existing tools and platforms that developers use. Since the goal of the City of Things project is to explore what could be possible in a Smart City context, it is hard to define requirements and commit to specific technology choices early in the process. Container managers such as Kubernetes (k8s) [26] provide very flexible APIs to manage containerized services. However, dependency-management and collaboration between services require ad-hoc solutions that are prone to break. Service meshes can aid in this, but they require putting additional components such as proxies in the data path. This overhead is even more damning if they are only used to solve operations issues. Given the latency requirements of many IoT applications such as smart grids, this is not a good fit for Smart Cities projects. Finally, service meshes provide a general fabric for all services to communicate with each other, instead of configuring specific connections based on a topology model of the application.

Thus, given these limitations of the state of the art, the contribution of this work is to answer the following research questions.

RQ 3.1. *On an abstract level, what concepts enable modeling and automated management of dependencies between services?*

RQ 3.2. *To what extend does the state of the art support modeling and automated management of such dependencies in Kubernetes?*

RQ 3.3. *How can existing platforms be extended in order to support service relationships without hiding the underlying API of the platform to users and without adding extra components in the data path?*

RQ 3.4. *What is the orchestration overhead introduced by adding support for such relationships?*

Additionally, this research provides an open source proof of concept prototype relations orchestrator for Kubernetes. This paper starts off by exploring how related work addresses some of the operational challenges of running complex interconnected applications and services in Section 4.2. The smart cities use-case is explained in detail in Section 4.3. Section 4.4 provides a number of definitions concerning what it means to have a relationship between two services and Section 4.5 uses these definitions to pinpoint what is lacking in the state of the art Kubernetes orchestration tools. We propose “orcon”, a proof of concept orchestrator on top of Kubernetes that manages relationships between services in section 4.6. We evaluate whether this approach is viable by comparing the proof of concept with the Juju orchestrator and native Kubernetes tools in Section 4.7. Finally, Section 4.9 concludes this paper and explains how future work will continue this line of research.

4.2 Related Work

Although cloud models are often presented as the solution to many operational challenges, their creation requires considerable technical and architectural expertise [18]. However, some of this expertise is already being captured in the form of cloud computing patterns. Fehling et al. [15] propose a way to describe the deployment in an abstract way using patterns, and to translate these patterns into the actual deployment models, with the goal of significantly reducing the required knowledge to create cloud models. Martino et al. use automated reasoning to map between cloud agnostic and vendor dependent cloud patterns [29]. Unfortunately, these approaches are not applicable to scenarios where full access to the underlying cloud infrastructure is required because it is hidden by the abstractions of the structural models. This eliminates the possibility of a multilevel approach where deployments can be modified using both higher-level and lower-level concepts.

Container orchestration is another recent development aiming to solve operational challenges. Topics such as resource scheduling, load balancing, fault tolerance and autoscaling are supported in most state-of-the art orchestrators [11]. However, higher-level abstractions, and specifically, the concept of relationships and dependencies is much less widespread. As Burns et al. note, Google's decade of experience with container orchestration has shown that dependency management is an important issue but the perceived complexity of dependency-aware systems has hampered the adoption of such systems by mainstream container-management systems [9].

4.2.1 Cloud Models as an Abstraction

Cloud models have also been proposed as a way to decrease the complexity of managing complex cloud applications by using it as a simpler abstract representation. This has had some success in the area of big data processing, for example [1]. Bhattacharjee et al. continue on this line of thinking and propose CloudCAMP [4] for domain-specific modelling so that cloud applications and their dependencies can be modeled without the need for domain expertise. The authors show that providing a higher-level abstraction to model cloud applications indeed reduces the complexity and enhances the ease of use. However, it requires pre-made building blocks to provide the higher-level abstractions. TOSCA is a front-runner in the field of cloud modelling languages and is used in many domains to simplify operations, however, because of its versatility and popularity, the risk exists of proliferation of incompatible TOSCA dialects [2].

4.2.2 Mutating Cloud Models

Finding effective ways to mutate cloud models is important in order to enable customization and day-2 operations such as maintenance and patching of applications managed with models. Managing dependencies, configuring and re-configuring services all require changing and updating cloud models. Palesandro et al. propose Mantus [32] as an aspect-oriented approach for modifying TOSCA models. They introduce the TOSCA Manipulation Language as an "XSLT for TOSCA models",

making it possible to model day-2 operations as changes to a cloud model. Some of the solutions in this space come from the industry, with Kustomize [27] as a prominent example of a language to modify Kubernetes models. Some tools such as Helm [20] go one step further by adding features such as package management, lifecycle management and dependencies to Kubernetes models. However, these are heavily criticized even within the Kubernetes community for conflating too many concerns in one tool and solving none of the challenges particularly well [16].

4.2.3 Relationships in Models

In order to make it easier to modify and reuse cloud models, the majority of cloud modelling languages supports creating topologies, where a cloud application is composed of a number of self-contained entities connected by relationships [3]. This is particularly relevant in NFV environments. Chaining of heterogeneous functions is important to both NFV and IoT platforms, although it is still an open research challenge in 2019 according to Vaquero et al. [42]. The Juju cloud modelling language, for example, is being used to orchestrate 5G Virtual Network Function (VNF) services [13]. These relationships are also useful for more data flow-based workloads [38][17].

An important advantage of topology-based cloud modelling languages comes from their ability to reuse components by turning a monolithic cloud model into a set of loosely-connected interchangeable components [44][38][40]. This also supports enhanced collaboration between multiple parties, for example by function shipping [45] and can even support a “marketplace”-like ecosystem with off-the-shelf components. Furthermore, these topologies can be used in order to analyze application topologies, find common microservice architectural smells, and suggest refactorings, as shown by Brogi et al. [8].

4.2.4 Smart city service orchestration

Service orchestration in smart cities is a complex multi-faceted issue. Sivrikaya et al. tackle cross-domain service composition by proposing the ISCO framework [41], a multi-agent-based middleware framework, which builds on top of the JIAC agent platform [21]. Not addressed by ISCO, however, is the issue of composition of existing polyglot services running in container orchestration platforms such as Kubernetes. The bIoTope project aims to build an ecosystem to create ad hoc and loosely coupled information flows in a smart cities context [24]. It introduces several building blocks in order to enable standards-based open communication and proof of concept implementations of this framework as an alternative to the traditionally proprietary and vertically-oriented ecosystems. The lower-level concerns of modeling, deployment and reconfiguration of containerized services based on compositions are not in the scope of the bIoTope project, however. The SWITCH workbench [46] offers a solution for managing the entire life cycle of time-critical applications in general. Using TOSCA as a modeling language, it supports management of applications consisting of complex topologies of microservices. Although it supports deploying applications to Kubernetes, it hides the entire Kubernetes API behind the TOSCA abstractions, making it hard to integrate with the wider Kubernetes ecosystem.

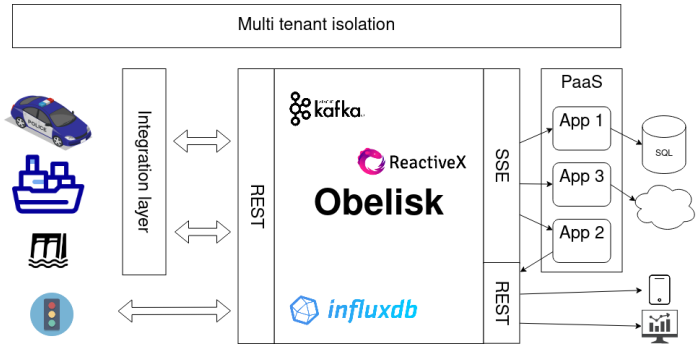

Figure 4.1: High level overview of the Obelisk City of Things architecture. Obelisk provides uniform and secure access to heterogeneous IoT data to multiple tenants with varying levels of cooperation. An in-house PaaS allows tenants to add additional processing functionality close to the data.

4.3 The Use-Case

As explained in the introduction, the City of Things project [34] is a collaboration between industry, academia and government with the goal of using IoT to make urban life more enjoyable and sustainable. As with any smart cities project, this creates a complex environment spanning multiple management domains that has to support multiple tenants with varying levels of collaboration between them. At the core of this setup, shown in Figure 4.1, sits Obelisk [7][22]; a platform for building scalable applications on IoT centric time series data. Obelisk is specifically built to support the heterogeneous nature of smart cities. Heterogeneity in protocols and sensors is supported by using a flexible REST interface and an integration layer capable of translating a wide variety of IoT protocols. Smart cities also introduce a second form of heterogeneity however, namely in terms of authorization, data access and data ownership. Since smart cities require collaboration between multiple parties who are direct competitors to each other, there are very stringent requirements on which data gets shared to which exact parties. Obelisk supports this using deeply integrated multi-tenant isolation with granular access controls in the entire architecture. The Obelisk and City of Things projects are explained in much more detail in previous work [7][34].

It became clear early in the project that there is a need for low-latency processing and transformation of the data captured by Obelisk. For this reason, the solution includes a multi-tenant Kubernetes-based Platform as a Service (PaaS) co-located with the Obelisk core. This allows customers to run event-based containerized applications that ingest event-based data streams from Obelisk using a Server-Sent Events (SSE) API, process them, and load them into either Obelisk or external platforms. Model-based management of these applications and their connections is the main focus of this paper. Figure 4.2 shows a simplified and zoomed-in view of the use-case where a number of different applications running inside of the Kubernetes PaaS connect to the SSE server. The *core team* develops and manages the Obelisk core, a number of different *app teams* develop applications using Obelisk's SSE API, and the *platform team* manages the Kubernetes cluster where

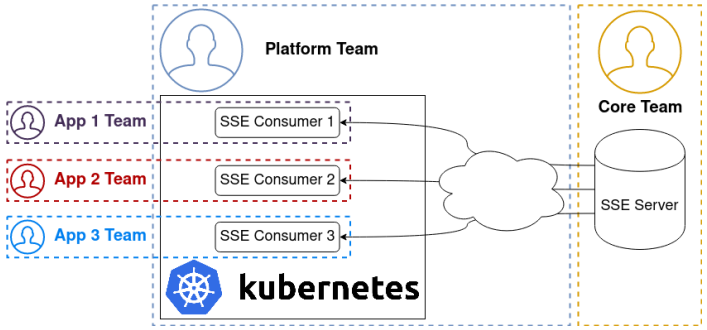

Figure 4.2: Different parts of the application run in different administrative domains, shown in the figure using dashed lines. Each third party team is a different tenant on the Kubernetes cluster. The platform team manages the cluster and its connection to external infrastructure. The core team manages an external SSE server.

the applications run. None of these parties has full knowledge and access to all the parts of the entire end-to-end application.

Automated model-based deployment and management of the applications is key here for a few reasons. First of all, it allows app teams to get started with the data as soon as possible and in an independent manner. Secondly, if the automation system propagates changes without human interaction, then all teams have the freedom to modify and iterate on their part of the software without coordination with other teams. Finally, because of the complexity of interconnections between different components, model-based management is key for its ability to retain a global view of the complete topology. This requirement is partly fulfilled by Kubernetes itself. It falls short, however, in modeling and managing the relationships between individual components. For example, it is not possible for Kubernetes to define that one service has a relationship to another, automatically configure the services depending on that relationship, and reconfigure the services when the service on the other end of the relationship changes. Moreover, the heterogeneity of stakeholders in a smart cities context adds additional challenges to service orchestration. First of all, it is very hard to standardize on a single methodology and toolset to deploy and manage applications. Although each application has components running in Kubernetes, these are often deployed and managed differently, depending on the stakeholder. Secondly, it is not possible to have a single stakeholder deploy and manage the entire end-to-end application as a single model because each application crosses multiple management domains.

Due to the technically challenging nature of this project, developers often use advanced features of Kubernetes in order to fine-tune how the applications are managed, scaled and upgraded. This includes integration with many tools in the Kubernetes ecosystem such as service meshes, custom resources and operators. Many of these tools either extend the Kubernetes API with new functionality or use the Kubernetes API to manage and update the application models. As such, using these tools requires direct access to the Kubernetes API and the model of the application. It is thus vital

that any solution does not impede the developer's access to the API so they can continue benefiting from the wider Kubernetes ecosystem. Similarly, the solution itself should also integrate into Kubernetes itself in such a way that existing management tools and workflows seamlessly integrate, showing the need for a Kubernetes-native solution.

4.4 Concepts of a service relationship

This section provides a definition and an extensive explanation of a service relationship and related concepts. The purpose of this is two-fold. Firstly, this explanation is used throughout this research to evaluate the state of the art. Secondly, this chapter forms a blueprint for how to fully support service relationships in orchestration systems, answering RQ 3.1. The section starts with the definition and proceeds to explain each part of the definition in detail. The terms in **bold** are used further in this work to refer back to specific parts of the definition.

Definition 4.1. *A service relationship is an explicit typed connection between isolated and independent service models that enables exchange of configuration information, synchronization of lifecycles and runtime communication.*

In the most simple sense, a relationship means that two services are connected with each other. This connection can have up to three distinct components.

1. The **communication component** refers to the interaction of the services at run-time. *Example: The SSE client communicates with the server using the HTTP SSE protocol.*
2. The **lifecycle component** refers to how the lifecycles of related services are dependent on each other. *Example: The SSE client can only start after the SSE server has started.*
3. The **configuration component** refers to how the configuration of one service uses information from another service. *Example: the SSE client app is configured with the URL of the SSE server.*

Each relationship has one or more of these components. The relationship between a web service using an SSL certificate and the certificate authority (CA), for example, has a lifecycle and a configuration component: the web service can only start after the CA is available and the web service is configured to use a certificate signed by the CA. This relationship does not have a communication component, however, since there is no communication between the webserver and the CA at runtime.

Relationships are **explicit** in the sense that they are defined in the model, rather than inferred from service configuration or runtime behavior. This important property makes sure operators have a clear view on the topology of their application, and automation tools have a straightforward way to reason about it.

Definition 4.2. *An interface is the directional **type** of a service relationship, describing the supported exchange of information and coordination between services.*

The interface specifies which components a relationship has and what each component entails. It defines the following.

- How *the lifecycles* of the two services are connected.
- *The protocol* used for runtime communication between the services.
- *What* configuration information is *shared* between the services.
- *How* configuration information is *presented* to each service.

An interface is directional in the sense that the service on each end of a relationship acts differently. As a result of this, an interface can be broken down into two sides: one for each service.

The type of a relationship is defined in an interface: it specifies which components a relationship has and what each component entails. It defines the following.

Definition 4.3. *A role is the part of an interface describing the supported relationship behavior of a single service.*

Two roles exist in each relationship: the *provider* and the *consumer*. One service *provides* the interface while the other service *consumes* it. A relationship is only possible between a service that supports the *provides* role and one that supports the *consumes* role of the same interface.

Relationships are created between **isolated** services in the sense that model and state are service-scoped. By default, information in the service scope is not available to other services and cannot be referenced in their models. Information can only be made available to related services by including it in a role. This behavior ensures all service relationships are explicit and completely described by the interface.

Furthermore, each related service is **independent**, meaning each service model can be managed as an independent entity, by an independent entity. This makes it possible for service relationships to cross administrative boundaries forming the basis for collaboration between independent parties.

4.5 Relationship Support in Kubernetes

This section evaluates the current support for relationships in Kubernetes based on the definitions of the previous section. Table 4.1 shows a summary of this evaluation, presenting the answer to RQ 3.2.

4.5.1 Native Kubernetes

In Kubernetes, the desired state of a cloud application is modeled using *object specs*. Although it is possible for two services to communicate with each other, these connections are not explicit in the object specs. The object spec specifies a unique name for each service. All containers in the same namespace as the service can use this name to establish runtime communication. In order to

	K8s	Helm	Service Mesh	Juju
Relationship Support				
Explicit	×	✓	×	✓
Typed	×	×	×	✓
Isolated	×	×	×	✓
Independent	×	×	×	✓
Communication comp.	✓	✓	✓	✓
Lifecycle comp.	×	×	×	✓
Configuration comp.	×	✓	×	✓
Kubernetes Nativity				
K8s API access	✓	✓	✓	×
Part of K8s API	✓	×	✓	×

Table 4.1: Comparison of relationship support in Kubernetes solutions. Although TOSCA-based solutions like Juju provide the full functionality of service relationships, they fall short in allowing users access to the full functionality of Kubernetes.

know which services will actually establish this communication, an operator would have to trace network traffic and/or inspect the code and declarations of every single service to see which ones initiate communication, as proposed by Muntoni et al. [31]. Although this functionality enables the communication component of a relationship, it does not support a lifecycle component nor a configuration component. Moreover, relationships are not explicit in the model and there is no notion of interfaces or relationship scope.

4.5.2 Helm

Helm is a package manager for Kubernetes. The desired state of an application is modeled in a Helm chart; a combination of templated Kubernetes object specs. Helm provides tools to fill in these templates, deploy the resulting objects and manage their lifecycle. Because these templates are based on Kubernetes object specs, Helm users can take full advantage of the Kubernetes API. Despite that, Helm itself is not part of the Kubernetes API. As a result, tools built to manage Kubernetes applications cannot take advantage of Helm.

Helm makes it possible for a chart to explicitly define its dependencies by specifying subcharts. As the name implies, this is an inherently hierarchical relationship requiring the subchart to be deployed as part of its parent, breaking the independence requirement stated in the previous chapter. As a result, this approach requires an operator who has complete authority over the entire model spanning all connected services. This does not allow creating services that span administrative boundaries as explained in Section 4.3.

Even though relationships in Helm are explicit, they are not typed. Although the model explicitly states *which* services are connected, it does not specify *how* they are connected. An operator needs to “reverse-engineer” this information by inspecting the template and checking whether services

are configured to communicate with each other, and inspecting the services themselves to see if they have a hard-coded connection to another service. Furthermore, Helm dependencies only provide one-directional isolation: a parent chart can override any values of the subchart while a subchart has no access to the parent chart. Lastly, these dependencies do not influence the lifecycles of the services. When Helm deploys a service, it does not wait until the dependencies of that service are running.

4.5.3 Service Meshes

A service mesh is a relatively new concept that seeks to improve communication between services by providing observability, increased security and failure recovery for requests between services. Istio [23], for example, implements a service mesh to connect containers running in Kubernetes. Conceptually speaking, service meshes aim to solve problems of the communication component of relationships, but they themselves cannot cover other aspects of a relationship between services because they provide no way to influence the lifecycle of connected services nor can they change the configuration of different services. Although service meshes can be useful to discover the topology of a microservice application, they infer this from the runtime behavior of services, instead of any explicit relationship definition. Furthermore, it is up to the administrator to make sure that connected services are actually compatible since service meshes do not have a way to declare and check the type of relationships.

A number of distributed tracing and observability solutions, such as Dynatrace [14], exist for Kubernetes. Much like service meshes, these applications make it possible to intelligently infer dependencies from the runtime behavior of microservice applications and present this dependency information to users as a topology model. Much like service meshes, however, these also suffer from the same issues: they provide no way to manage the lifecycle and configuration of services based on a topology model. In a sense, it's the reverse of our goal: instead of changing run-time behavior to conform with a model, it creates a model based on run-time behavior.

Another, more practical issue specific to service meshes is that these are often implemented using sidecar proxies. These add latency and increase the resulting complexity of a deployment. Moreover, these proxies often only support a limited number of communication protocols.

Although service meshes and distributed tracing solutions are a useful development, they do not provide any additional features over Kubernetes in terms of service relationships as defined in Section 4.4. For this reason, we see them as complementary to service relationships. Thus, the remainder of this research does not regard these as an alternative solutions but evaluates whether different solutions can integrate with them by determining whether the model allows direct access to the Kubernetes API.

4.5.4 TOSCA-related solutions

A number of different initiatives are working towards TOSCA-based solutions to manage applications running on top of Kubernetes. Projects with an industry background such as Juju and Cloudify provide full-featured orchestrators with Kubernetes support. This integration also has considerable interest from academia with a number of recent publications such as Chareonsuk et al., proposing a TOSCA to Kubernetes translator [12], Bogo et al. introducing a toolchain for deploying multicomponent applications using TOSCA [5], and Borisova et al. examining how to adapt TOSCA for Kubernetes deployment [6].

Because of TOSCA's exhaustive support for relationships, all these tools support explicit and typed modelling of communication[30], lifecycle [39] and configuration components of a service relationship. Moreover, individual components in TOSCA are isolated and a number of TOSCA implementations, such as Juju, support creating relationships between completely independent models. As a result, it is possible in Juju to create relationships crossing administrative boundaries, as required by the use-case explained in Section 4.3. Even though most of these TOSCA-based solutions have complete support for service relationships as defined in Section 4.4, there are two issues that make them unsuitable for our use-case. The first one is that it is often an all-or-nothing approach: taking full advantage of this relationship functionality is only possible if the entire application is modeled using the TOSCA-based platform. But this is not always feasible, as our use-case shows: many collaborators use their own tools and methodologies to manage their infrastructure and are hesitant to change. The second issue is that TOSCA adds an additional abstraction *on top of* Kubernetes, which hides Kubernetes itself. Such abstractions, when done well, can simplify complex platforms but they have the downside that they often do not support all the features of Kubernetes itself and that they make it hard to reuse existing Kubernetes tools and expertise. Next to this, it also makes migrating to the new abstraction more difficult because there is no a clear migration path available and it is difficult to gradually transition to the new abstraction. For the remainder of this paper, we will use Juju as a representative TOSCA-based solution.

4.6 Implementation

This section presents the open source *orcon* orchestrator [37] developed as part of this research. The motivation behind the development is three-fold. First of all, this implementation allows us to check the validity of the concepts and definition of service relationships presented in Section 4.4 and determine whether these are a sufficient answer to RQ 3.1. Secondly, this implementation shows how to extend an existing platform to support service relationships without hiding the underlying platform API, answering RQ 3.3. Finally, this implementation is used to answer RQ 3.4 by evaluating its performance in Section 4.7.

The orcon orchestrator injects the concepts of *relationships*, *interfaces* and *roles* into the Kubernetes API. By using injection, orcon avoids creating an additional layer of indirection and enables users to keep working with the same tools and techniques they are used to.

Kubernetes works based on the desired state principle: users declaratively describe the desired state of the system by adding and changing *object specs* in the Kubernetes *API server*. Kubernetes services then take appropriate action to get the system into that state and reflect the current state in the *object status*. The object status thus describes what is actually set up in the cluster in order to meet the needs described in the object spec. Orcon allows users to describe desired relationships between objects by adding additional information to the object specs. Two orcon services monitor the API server for these descriptions and take the appropriate actions to establish the desired relationships. Since modifying object specs is already supported by means of the Kubernetes API, all Kubernetes tools that support this API automatically support orcon too.

The next subsection explains how the conceptual model of a relationship from Section 4.4 is implemented in the existing Kubernetes API in order to allow users to describe desired relationships between objects. The remaining subsections explain how the orcon services extend the Kubernetes control plane in order to establish those relationships.

4.6.1 Representing relationships, interfaces and roles in Kubernetes objects

Orcon adds a number of extensions to the schema of Kubernetes object specs that are directly mapped to the conceptual model of a relationship detailed in Section 4.4. For these extensions, orcon uses *Annotations*, which allow adding arbitrary complex metadata to any object, and *Custom Resources*, which allows adding new types of objects to the Kubernetes API.

The **roles and interfaces** supported by an object are described using the *orcon.dev/provides* and *orcon.dev/consumes* annotations. Each annotation contains a comma-separated list of interface names of which the object supports that specific role. These annotations are used by orcon in order to type-check relationships and in order to figure out the interface of a relationship between two objects.

The relationships themselves are described by adding the annotation *orcon.dev/relationship* to the *consumer* end of a relationship. It supports a comma-separated list of object names so that a single object can have multiple relationships. Each object name in this list specifies the request for an individual relationship between the object in question and the named object. Note that in the Kubernetes model of "desired state", these annotations denote the *desire* for a relationship between two objects, not necessarily an *established* relationship. Establishing a relationship is only possible if the specifying object supports the *consumer* role of an interface that the named object *provides*.

These three annotations are enough to describe the intent for a basic relationship between a Kubernetes Deployment and a Kubernetes Service. Below is an example of a Deployment consuming the *sse* and *mysql* interface, which has a relationship to an object named *sse-endpoint* and an object named *mysql-db*.

```
kind: Deployment
metadata:
  name: sleep
  annotations:
    orcon.example/consumes: sse,mysql
    orcon.example/relations: sse-endpoint,mysql-db
spec:
  ...
```

In such a basic relationship, the interface is inferred implicitly based on the default orcon relationship template and the name of the interface. The default interface template is as follows.

- The *lifecycles* of the two objects are connected in such a way that the Deployment starts after the service becomes available.
- Orcon assumes both objects use the same *protocol* for communication.
- The providing Service *shares* its URL.
- The service URL is presented to the consuming Deployment as an environment variable with the same name as the interface itself.

Since such an interface is very limited in its usefulness and provides only rudimentary type-checking, orcon allows users to explicitly define interfaces by specifying both roles of an interface using two custom resources: *ProviderConfig* and *ConsumerConfig*.

The **ProviderConfig** explains how to enact the *provider* role of an interface. It defines which values should be extracted from the providing object. The *valueLocations* map in a ProviderConfig object describes for each relationship key, where to extract the associated value from. These values can come from the object spec, from the object state, a Secret or a ConfigMap.

Below is an example of a ProviderConfig mapping three relationship keys to a field in the object spec and two secrets.

```
apiVersion: relations.orcon.example/v1alpha1
kind: ProviderConfig
metadata:
  name: mysql-config
config:
  valueLocations:
    url:
      type: /v1/services
      name: mysql-service
      path: spec.externalName
    username:
      type: secret
      name: mysql-secret
    password:
      type: secret
      name: mysql-secret
```

Important to note here is that, using this method, a providing service does not actually have to run inside of the Kubernetes cluster itself. The only requirements is that a *representation* of the

service is present in the API server in the form of an object. The above example uses the *externalName* functionality of Kubernetes Service objects, which allows representing external services in Kubernetes objects.

The **ConsumerConfig** explains how to enact the *consumer* role of the relationship. It defines how the relationship values should be injected into the consuming object and what kind of lifecycle dependency the consuming object has on the providing object.

- The *lifecycledep* key describes the lifecycle dependency between the provider and the consumer of a relationship. At the moment, the only supported lifecycle dependency is *start*, denoting that the consuming service needs to start *after* the providing service. This field also accepts the string *none*, denoting there is no lifecycle dependency.
- The *keyconfig* map describes for each relationship key, how to inject this key into the consuming object using *injectionMethod*. Orcon currently supports injecting relationship values using environment variables and mounted volumes.

Below is an example of a ConsumerConfig mapping three relationship keys to two environment variables and a volume. It also describes that the object should only start after the related object has started.

```
apiVersion: relations.orcon.example/v1alpha1
kind: ConsumerConfig
metadata:
  name: mysql-consumer-config
config:
  lifecycledep: start
  keyConfig:
    url:
      targetKeyName: mysqlurl
      injectionMethod: env
    username:
      targetKeyName: mysqlusername
      injectionMethod: env
    password:
      targetKeyName: pwd
      injectionMethod: volume
      mountPath: /etc/mysql
```

Finally, objects specify which configuration they use for a certain interface with the optional *orcon.dev/config* annotation.

4.6.2 Injecting relation data

Figure 4.3 shows the architecture of the orcon services responsible for taking the appropriate *actions* to establish and manage relationships. The extraction and injection of relation data is managed by the Relations Controller. This service implements the Kubernetes Controller pattern [19][25] in order to plug into the Kubernetes management plane. Controllers are Docker

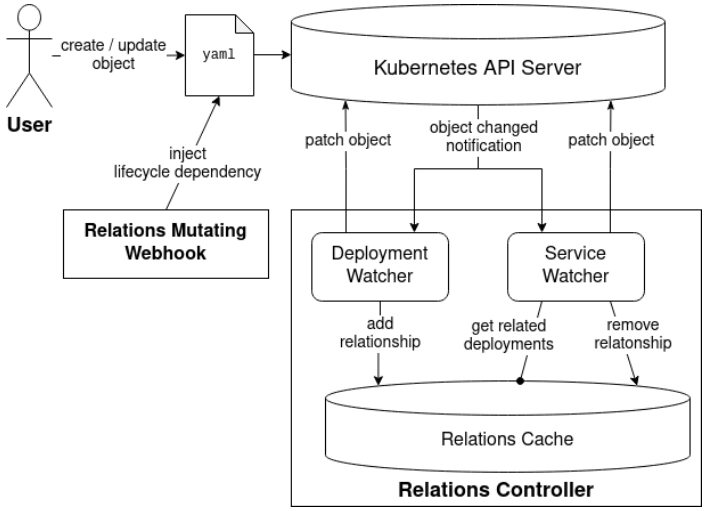

Figure 4.3: Architectural overview of orcon. The Relations Mutating Webhook injects lifecycle dependencies before the objects are persisted in the API server. The Relations Controller modifies Deployments and Services in order to establish and update requested relationships.

containers running inside of the Kubernetes cluster that use the Kubernetes API to observe and modify resources.

The relations controller consists of three parts: A Deployment Watcher, a Service Watcher and a Relations Cache. The Relations Cache maps the names of providing objects to objects that request a relationship to them. The sequence diagram in Figure 4.4 shows that the Deployment Watcher updates the Relations Cache every time a Deployment gets added. The Relations Cache is then used by the Service Watcher to figure out which Deployments to update when a Service changes. The ProviderConfig and ConsumerConfig are used to determine which specific actions to take and how to update related objects. These watchers perform very similar functionality when Deployments and Services change after creation.

4.6.3 Injecting lifecycle dependencies

The orcon Relations Mutating Webhook is responsible for injecting lifecycle dependencies into Deployments as shown in Figure 4.3. This service implements the Mutating Admission Controller pattern [19][33] in order to have the ability to change Deployments *before* they are persisted in the API server. Thus, the lifecycle dependencies are injected before the Kubernetes services responsible for deploying Pods can view them. This avoids a race condition where Kubernetes deploys Pods before the lifecycle dependencies are injected.

The lifecycle dependencies themselves take the form of Kubernetes init containers that wait until they receive a signal that the dependent object has started. Since the main container of a Pod will

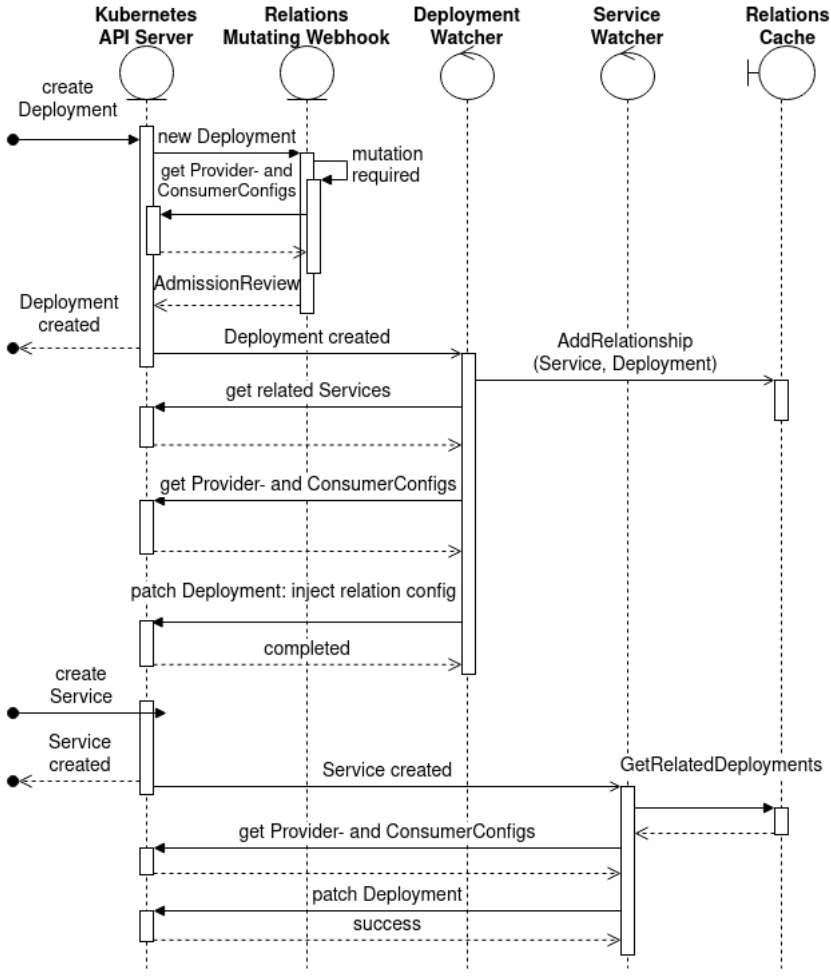

Figure 4.4: The Relations Mutating Webhook injects lifecycle dependencies before the objects are persisted in the API server. The Relations Controller modifies Deployments and Services in order to establish and update requested relationships.

only run after all init containers have exited, adding such an init container effectively halts the execution of the main container until the lifecycle dependency is met. This way, orcon makes sure that all objects start in the correct order.

Figure 4.4 shows the sequence of the *orcon* admission controller when a new Deployment is created. The controller checks if the Deployment consumes an interface. If so, the controller injects the *orcon* Init Container into the pod template of the Deployment and configures it according to the requirements of the interface. As a result, the application containers of that Deployment will only start after the relation information is added by the relations controller. Important to note here is that the relations controller will only spring into action when a valid relationship is requested and is possible between two objects. The admission controller, however, will inject the lifecycle dependencies immediately, even if the requested relationship is not possible. This ensures that, in the event the providing object is not yet present in the API server, the consuming object will not start. If an object has a lifecycle dependency on another object, it should not start if that other object is not present.

Although it is technically possible for the Admission Controller to inject relationship data, orcon avoids it to reduce latency because Admission Controllers block the acceptance of an object until they are finished. Regular controllers, on the other hand, work concurrently with other operations on those objects thanks to Kubernetes' optimistic concurrency control [9][35].

4.6.4 Optimization

As explained in Section 4.6.1, orcon heavily uses Kubernetes object annotations to store metadata. Since annotation contents are not indexed by the Kubernetes API, it is not possible to select objects based on them. In order to find all objects which consume a certain relationship, orcon needs to request all objects having any relationship and manually search through the annotations itself.

The first step in reducing the overhead of this process is to cache information of related objects locally in the controller so the Kubernetes API does not have to be contacted in order to retrieve information. For this, orcon uses the SharedIndexInformer of the Kubernetes controller SDK. By accessing this eventually-consistent cache directly, orcon avoids expensive calls to the Kubernetes API. Since the Kubernetes API itself is also an eventually-consistent system, orcon natively supports this paradigm without modifications.

The second step in reducing the overhead of object annotations is the RelationsCache. This orcon-specific data structure maps the names of providing objects to cached versions of all objects which consume a relationship with them. This way, when a providing object is updated, finding all objects to whom the change needs to be propagated is an O(1) operation which happens in the controller itself without contacting the Kubernetes API. Every time a relationship changes, orcon updates the RelationsCache to reflect those changes.

The controller itself currently does not support any parallelism. All updates to Kubernetes objects are processed sequentially, one by one.

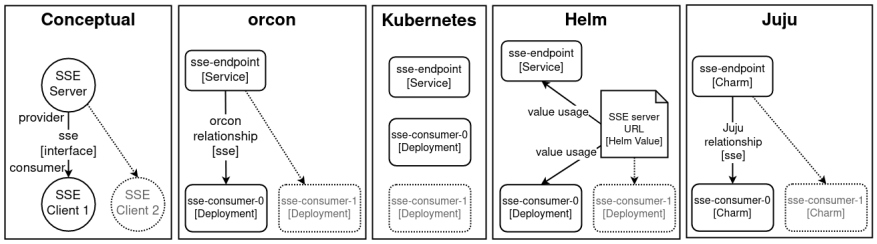

Figure 4.5: Overview of the conceptual topology of the evaluated use-case and the resulting topologies of its implementation using each evaluated solution. The dotted graphics show the components needed to add an additional consumer.

4.7 Evaluation

This evaluation compares the performance of the *orcon* orchestrator presented in the previous section to regular Kubernetes and to Juju on Kubernetes, in order to answer RQ 3.4.

The leftmost part of Figure 4.5 shows the evaluated conceptual topology. This test case is based on the use-case described in Section 4.3: a number of separate app teams each provide a single application that runs on the Kubernetes cluster managed by the platform team and connects to a single eternal SSE server managed by the core team. The gray and dotted parts of Figure 4.5 show which components are added with each additional app team.

The *sse* relationship in this topology has the following components:

- **Communication:** The SSE client connects to the SSE server to receive and process events.
- **Lifecycle:** The SSE client can only start *after* the SSE server has started.
- **Configuration:** The SSE client receives the DNS name of the SSE server.

Although orcon supports much more complex relationships and topologies, this evaluated use-case is intentionally simplified for clarity purposes.

4.7.1 Setup

All performance benchmarks are executed on a vanilla Kubernetes cluster from the Charmed Distribution of Kubernetes (CDK) version 1.14.1 [10] connected to a Ceph cluster for persistent storage. All software is deployed in virtual machines on a VMWare ESXI cluster [43] and managed by Juju 2.5.4 [28]. Each solution is tested with an increasing number of consumers, from 5 to 55 with an increment of 5. Each combination is tested 20 times. The graphs show all measurements as individual dots and crosses. The full source code for the different implementations, the evaluation and the full specification of the test cluster is available on GitHub [36].

4.7.2 Evaluated Solutions

These evaluations compare four different solutions to the use-case described in Section 4.3.

1. The "*orcon*" solution deploys the consumers as described in Section 4.6.
2. The "pure k8s" solution deploys the consumer containers by submitting a `deployment.yaml` using the `kubectl` command-line client. The URL is specified using a `ConfigMap`. The URL is updated by submitting a new `deployment.yaml` file with the updated URL.
3. The "Helm" solution deploys the same setup as the "pure k8s" solution but the Deployment is templated so that the number of consumers and the URL are specified using Values.
4. The "Juju" solution deploys the consumers using Kubernetes Charms. Each Consumer is a k8s charm that deploys the consumer container and the SSE service is represented by a proxy charm that contains the URL to the SSE service. This URL is transferred to the consumer charms using a Juju relationship. The URL is updated by changing the configuration of the SSE service charm, which then sends the updated URL to all the consumer charms, which in their turn update the PodSpec.

Figure 4.5 shows the evaluated conceptual topology and the resulting implementation in each solution. The specific models and implementations of each solution are available on GitHub [36].

Note: even though Juju is used to manage the Kubernetes cluster itself, only the "Juju" solution uses Juju for the deployment of the consumers. The other solutions simply deploy on top of the Juju-managed Kubernetes clusters.

4.7.3 Functional Evaluation

With the goal of comparing the functionality of orcon to the state of the art, we used BPMN 2.0 choreography diagrams to model the interactions required to deploy and update the aforementioned setups. For clarity purposes, interactions where at least one party is a person have a solid border, interactions where both parties are people have an icon in the description, and interactions solely between software systems have a dashed border. We will mainly focus this evaluation on interactions involving humans since those have a significant penalty in terms of latency and potential for mistakes. The performance of the machine-to-machine interactions are benchmarked in Section 4.7.4.

Figure 4.6 shows the processes required to create a new app in each solution. These processes assume the appropriate actions have already been taken to add the SSE server to the setup. Helm has the significant downside that it requires the platform team to submit the application on behalf of the app team. Due to Helm's lack of independence in relationships, the entire setup, including the service and existing applications from other teams, needs to be managed as a single entity and only the platform team has the permissions to do this.

Juju has the downside that the app team cannot interact with Kubernetes directly, Juju serves as

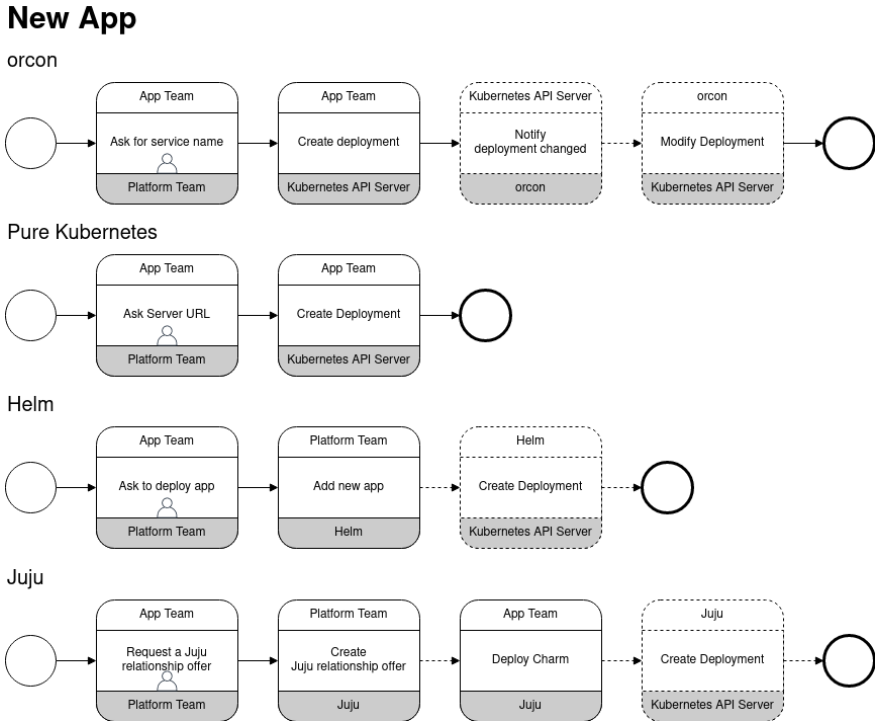

Figure 4.6: Like any TOSCA-based solution, Juju has the downside that users cannot interact with Kubernetes directly, Juju serves as an intermediary.

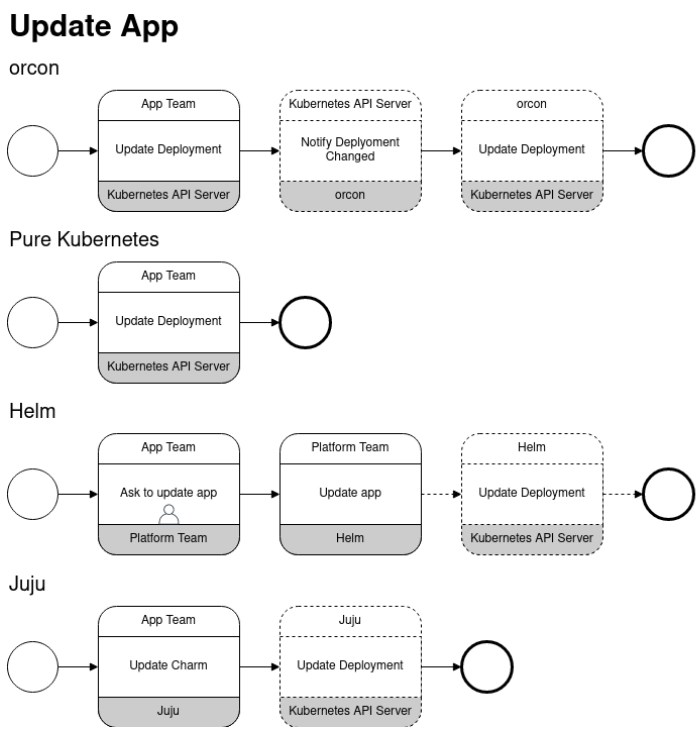

Figure 4.7: The Helm setup requires the platform team to change an App because it manages the entire setup as a single entity.

an intermediary. This shows the advantage of the more integrated approach of orcon: it adds additional abstractions without an additional abstraction layer. Although the Juju solution requires three manual interactions instead of two, this is simply due to Juju's mandatory security concerning relationships that cross management domains. This difference would not exist if all solutions provided the same level of security.

Figure 4.7 shows the processes required to update an existing app. Here too, Helm suffers from its lack of independence: the app team needs to ask the platform team to update the app on their behalf. Not only does this prevent the app team from interacting with Kubernetes directly, it adds an additional manual step in the process. Juju also has the same downside that its additional abstraction layer prevents the app team from interacting with Kubernetes directly.

Figure 4.8 shows the role relationships can play in automated response to changes. In orcon and Juju, only the core team has to perform a manual interaction to update the service. The system then automatically propagates changes to the related Deployments. While Helm dependencies could offer a similar function, it still requires an additional manual action because only the platform team has the permissions required to use it. Juju still has the same downside that the Kuber-

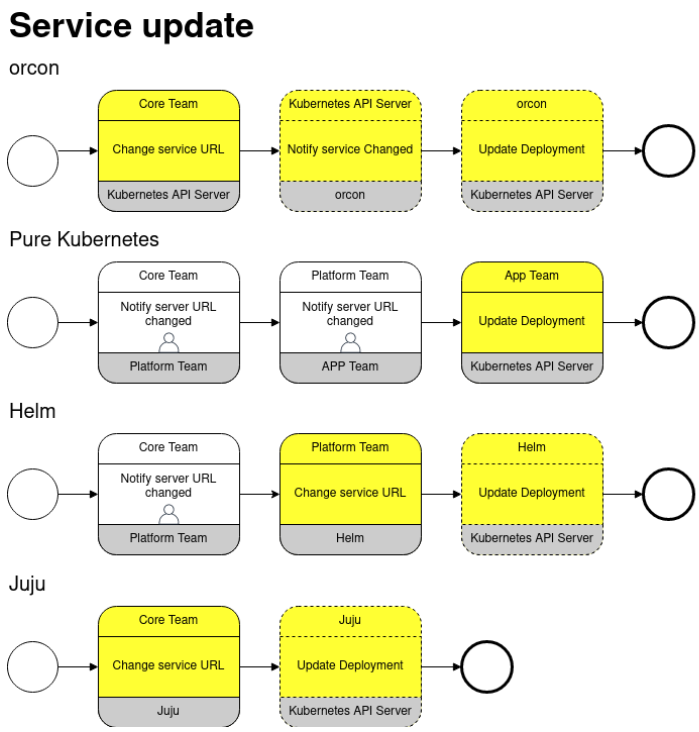

Figure 4.8: With the orcon and Juju setup, the relationship causes the service change to propagate automatically to connected apps without human interaction. The yellow interactions in these diagrams are benchmarked in the performance evaluation.

netes API is hidden from the users. The Pure Kubernetes setup requires the most manual actions due to having no automated way to propagate changes.

4.7.4 Performance Evaluation

With the goal of investigating the overhead of orcon, we benchmarked the time it takes, after a service is updated, to propagate that change to all Deployments. Note, however, that this benchmark does not take into account steps that require human-to-human interaction, so for the "helm" and "pure k8s" solutions not all steps required to update a service are benchmarked. The steps included in this benchmark are highlighted in yellow in Figure 4.8.

The *orcon* solution proposed in this paper propagates the URL change substantially quicker than Juju. As Figure 4.9 shows, *orcon* propagates the change to 55 consumers in 48 seconds on average, while Juju requires 146 seconds. The "Juju agents" plot in this graph shows how long it takes for the Juju agents to become ready to process a new change. There is a period of 100 seconds between when Juju updates 55 consumers and when the agents are ready to accept new changes.

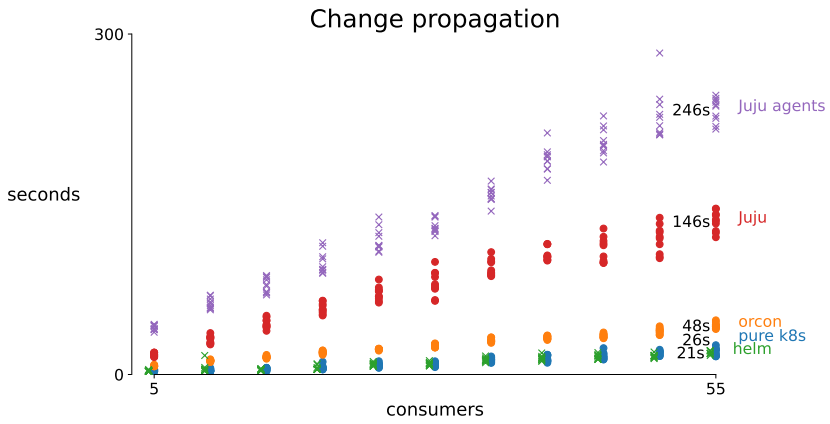

Figure 4.9: The *orcon* orchestrator proposed in this paper is significantly faster than Juju in propagating the new URL of the SSE server to the SSE consumers. Moreover, after all consumers are updated, *orcon* is immediately ready to accept new changes while Juju requires a cooldown period during which it cannot propagate URL changes.

This is because, for stateless services, the Juju management agent lifecycle is connected to the pod lifecycle. This means that each time a management agent updates a pod, both the pod and the management agent itself shut down and are replaced. As a result, there is a long period after the consumers are updated where the new Juju agent cannot accept new changes because it is initializing. The pure k8s and *orcon* solutions do not have such a cooldown period: these immediately accept new changes after updating the consumers.

Figure 4.10 dives deeper into the difference between our *orcon* approach, Helm and pure k8s. Since the pure k8s and Helm solutions do not have lifecycle management, the graph also includes a plot of change propagation duration of *orcon* without lifecycle management labelled “orcon without initc”. This shows that for 55 consumers, the average additional overhead of lifecycle management using init containers is nine seconds. Although it appears from this graph that the Helm and “pure k8s” solutions are significantly faster than orcon, this does not take into account the manual steps that require human-to-human interactions. These interactions are error-prone and introduce a highly-variable latency that can easily exceed the less than thirty seconds delay between orcon and the state of the art.

Figure 4.10 also shows that Helm, on average, performs slightly better than the “pure k8s” solution. There are a number of possible explanation for this behavior. The “pure k8s” solution uses a simple method to resubmit the entire application, including all the components that did not change, directly to the Kubernetes API. Helm, on the other hand, has intimate knowledge about what exactly changed due to the use of Helm Values. This might cause helm to interact with the Kubernetes API in a smarter way so as to only change the objects that are actually changed. This behavior was not investigated further because the main focus on this paper is on the performance

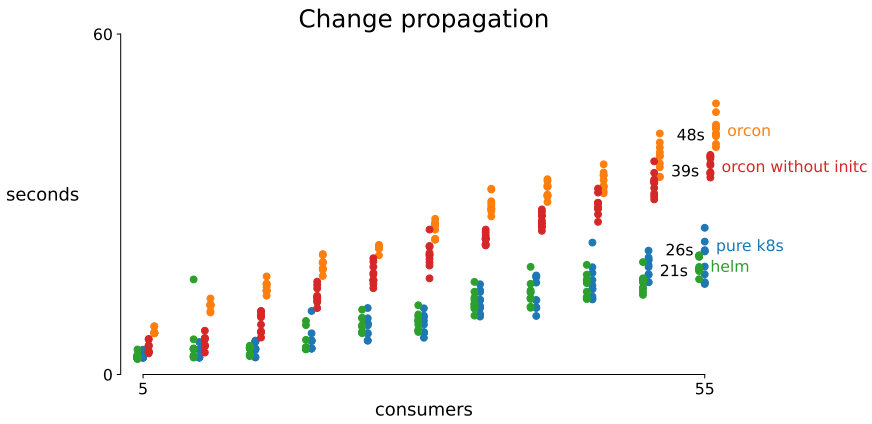

Figure 4.10: The init container used by *orcon* for lifecycle management, adds on average an additional 9 seconds to the time for a URL change in the SSE server to be propagated to all consumers.

of orcon compared to the state of the art. Performance between the different state of the art solutions themselves is of less significance to this research.

4.7.5 Summary

Table 4.2 shows a summarized comparison of orcon with the state of the art. For every process, orcon and Juju have the lowest number of human-to-human interactions required. Although Helm succeeds in reducing the number of human-to-human interactions needed to update a service, it still requires one such interaction because of Helm's monolithic approach to dependencies.

Orcon has an order of magnitude less overhead compared to Juju. Although orcon appears to have a slight overhead of less than half a second per pod compared to Helm, this is negated by the previously shown fact that helm still requires a human-to-human interaction for this process.

Although Juju makes dependencies explicit and allows the app team to update their service independently of other teams, it adds an additional abstraction layer that makes it impossible to use the full power of the Kubernetes API. Because Juju and Helm objects are not modeled in the Kubernetes API server itself, it is not possible to use other Kubernetes tools to create and manage these objects. Orcon, on the other hand, allows users to directly access the Kubernetes API server and is completely implemented inside of it. The power of this last feature is shown by the fact that other Kubernetes tools, including Helm, can be used to interact with orcon. Orcon is thus not strictly a competitor to Helm, since orcon can be used to enrich a Helm-based approach with true relationship-based dependencies.

Solution	**orcon**	K8s	Helm	Juju
Workflow: number of manual interactions				
New app	1	1	1	1
Update app	0	0	1	0
Update Service	0	2	1	0
Performance: overhead compared to K8s (s/pod)				
Change propagation	0.44	/	0	4.4
Kubernetes Nativity				
K8s API access	✓	✓	✓	×
Part of K8s API	✓	✓	×	×
Relationship support				
Explicit	✓	×	✓	✓
Typed	✓	×	×	✓
Isolated	✓	×	×	✓
Independent	✓	×	×	✓
Communication comp.	✓	✓	✓	✓
Lifecycle comp.	✓	×	×	✓
Configuration comp.	✓	×	✓	✓

Table 4.2: The evaluation shows orcon provides all the benefits of service relationships on Kubernetes while completely integrating into the Kubernetes ecosystem and providing much better performance than Juju.

4.8 Discussion

RQ 3.1 asks *"On an abstract level, what concepts enable modeling and automated management of dependencies between services"*

These concepts are laid out in Section 4.4, starting with the definition of service relationships: *"An explicit typed connection between isolated and independent service models that enables exchange of configuration information, synchronization of lifecycles and runtime communication"*. Furthermore, the concepts "interface" and "role" described in that section are required to model the full extent of both active and possible dependencies between services in a way that both humans and machines can easily understand and reason with them. By implementing these concepts in orcon and evaluating its functionality, we show these concepts indeed make it possible to model a service relationship and take full advantage of its benefits.

RQ 3.2 asks *"To what extend does the state of the art support modeling and automated management of such dependencies in Kubernetes?"*

Using the concepts of the previous answer, we evaluated the state of the art in Section 4.5 and came to the conclusion that, although TOSCA-based solutions offer full support for service relationships on Kubernetes, they fail to allow users access to the underlying orchestrator.

RQ 3.3 asks *"How can existing platforms be extended in order to support service relationships without hiding the underlying API of the platform to users and without adding extra components*

in the data path?"

Section 4.6 implements *orcon* shows how to use the introduced concepts for implementing service relationship support while maintaining user access to the underlying orchestrator. The orchestrator does this by injecting additional abstractions into the Kubernetes API, instead of wrapping it. As a result, orcon users can still take full advantage of the Kubernetes API, and existing Kubernetes ecosystem tools can be used to drive orcon. The *orcon* framework actively resolves dependencies between services and automatically propagates changes in them. The evaluation in Section 4.7 includes a confirmation of this functionality and its advantages for developer workflows.

RQ 3.4 asks *"What is the orchestration overhead introduced by adding support for such relationships"*

Section 4.7 shows that, although adding these concepts adds a slight orchestration overhead of 0.44 seconds per consumer compared to manual configuration, it removes the need for manual human-to-human interactions, ultimately reducing the total time needed to update services. Moreover, the overhead of orcon is an ten times smaller than that of Juju. This evaluation also show that resolving lifecycle dependencies at the container orchestration level also adds additional overhead. It is thus advised to modify the services to resolve their own lifecycle dependencies at runtime. This has the added benefit that it makes the services more resilient to dependencies breaking after the initial deployment.

Although Juju has much more overhead compared to orcon, it is important to note its much broader feature-set. Juju supports automatic cross-cluster relationships, allows extensive modeling and management of services in and beyond Kubernetes and is network and storage-aware. In cases where orcon's deep integration within the Kubernetes ecosystem is not needed and standardization on a single management tool is possible, Juju can be considered a powerful alternative to add relationship support to Kubernetes. Interesting to note is that, according to our evaluation, about half of the overhead of Juju is caused by a single design decision, namely replacing management agents when the pods they manage restart. This suggests the performance differences between Juju and orcon might not be inherent to Juju's expanded feature-set.

4.9 Conclusion

This research proposes orcon, an orchestrator that adds native support of relationships to Kubernetes. It is the first orchestrator that does so without hiding the underlying API and integrating in a way that supports the existing ecosystem of kubernetes tools. Our evaluation shows orcon propagates change at an average of 0.44 seconds per service, an order of magnitude faster than the state of the art.

An interesting future research opportunity is to investigate the overhead difference between orcon and Helm to shed light on possible optimization routes. Another interesting route to explore is to save historical relationship data in order to support easy rollback to previous configurations. This

is not possible in the current version of orcon as relationship updates are destructive in the sense that they overwrite previous values. A third opportunity lies in support for Federated Kubernetes clusters. Although the current implementation technically allows creating a relationship to a service in another Kubernetes cluster using the *externalName* functionality explained in Section 4.6, this still requires manual modification of representative Service objects. An improvement in this area would allow completely automated management of application topologies spanning multiple Kubernetes clusters, opening the door for full topology-based management from the cloud to the edge. Finally, orcon currently only supports relationships between equal peers. Investigating support for hierarchical relationships is an interesting path forward as it could enable the creation of higher-level abstractions inside of the Kubernetes API.

Acknowledgment

The authors wish to thanks Lennart Onghena for his valuable contributions to orcon.

Bibliography

[1] C. A. Ardagna, V. Bellandi, P. Ceravolo, E. Damiani, M. Bezzi, and C. Hebert. A Model-Driven Methodology for Big Data Analytics-as-a-Service. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 105–112, June 2017. doi:10.1109/BigDataCongress.2017.23.

[2] Julian Bellendorf and Zoltán Adám Mann. Cloud Topology and Orchestration Using TOSCA: A Systematic Literature Review. In *ESOCC*, 2018. doi:10.1007/978-3-319-99819-0_16.

[3] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys*, 51(1):22:1–22:38, February 2018. doi:10.1145/3150227.

[4] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda. A Model-Driven Approach to Automate the Deployment and Management of Cloud Services. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 109–114, December 2018. doi:10.1109/UCC-Companion.2018.00043.

[5] 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(9):1793–1821, 2020. doi:10.1002/spe.2848.

[6] Alexandra Borisova, Valeriya Shvetcova, and Oleg Borisenko. Adaptation of the TOSCA standard model for the Kubernetes container environment. In *2020 Ivannikov Memorial Workshop (IVMEM)*, pages 9–14, September 2020. doi:10.1109/IVMEM51402.2020.00008.

[7] Vincent Bracke, Merlijn Sebrechts, Bart Moons, Jeroen Hoebeke, Filip De Turck, and Bruno Volckaert. Design and evaluation of a scalable Internet of Things backend for smart ports. *Software: Practice and Experience*, 51(7):1557–1579, 2021. doi:10.1002/spe.2973.

[8] Antonio Brogi, Davide Neri, and Jacopo Soldani. Freshening the Air in Microservices: Resolving Architectural Smells via Refactoring. In Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Y. Nakagawa, editors, *Service-Oriented Computing - ICSOC 2019 Workshops*, Lecture Notes in Computer Science, pages 17–29, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45989-5_2.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, January 2016. doi:10.1145/2898442.2898444.

[10] Canonical Ltd. Charmed Kubernetes | Juju, 2021. URL: https://jaas.ai/canonical-kubernetes.

[11] Emiliano Casalicchio. Container Orchestration: A Survey. In Antonio Puliafito and Kishor S. Trivedi, editors, *Systems Modeling: Methodologies and Tools*, EAI/Springer Innovations in Communication and Computing, pages 221–235. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-319-92378-9_14.

[12] Woramon Chareonsuk and Wiwat Vatanawood. Translating TOSCA Model to Kubernetes Objects. In *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 311–314, May 2021. doi:10.1109/ECTI-CON51831.2021.9454890.

[13] Enrique Chirivella-Perez, Jose M. Alcaraz Calero, Qi Wang, and Juan Gutiérrez-Aguado. Orchestration Architecture for Automatic Deployment of 5G Services from Bare Metal in Mobile Edge Computing Infrastructure. *Wireless Communications and Mobile Computing*, 2018. doi:10.1155/2018/5786936.

[14] Dynatrace LLC. Dynatrace | The leader in automatic and intelligent observability, 2021. URL: https://www.dynatrace.com/.

[15] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer-Verlag, Wien, 2014. URL: https://www.springer.com/gp/book/9783709115671.

[16] Brian Grant. [Public] Felloe, Spokes, Axel, and Tiller: A Look at the parts of Helm, February 2018. URL: https://docs.google.com/presentation/d/10dp4hKciccincnH6pAFf7t31s82iNvtt_mwhlUbeCDw/edit?usp=embed_facebook.

[17] Michael Hahn, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Modeling and Execution of Data-aware Choreographies: An Overview. *Comput. Sci.*, 33(3-4):329–340, August 2018. doi:10.1007/s00450-017-0387-y.

[18] Lukas Harzenetter, Uwe Breitenbücher, Michael Falkenthal, Jasmin Guth, Christoph Krieger, and Frank Leymann. Pattern-Based Deployment Models and Their Automatic Execution. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 41–52, December 2018. doi:10.1109/UCC.2018.00013.

[19] Michael Hausenblas and Stefan Schimanski. *Programming Kubernetes: Developing Cloud-Native Applications*. O'Reilly Media, Sebastopol, CA, 1st edition edition, August 2019.

[20] Helm Authors and The Linux Foundation. Helm, 2021. URL: https://helm.sh/.

[21] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging Agents and Services — the JIAC Agent Platform. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer US, Boston, MA, 2009. doi:10.1007/978-0-387-89299-3_5.

[22] IDLab (Ghent University, imec). Obelisk, 2021. URL: https://obelisk.ilabt.imec.be/api/v2/docs/.

[23] Istio Authors. Istio, 2021. URL: https://istio.io/latest/.

[24] Asad Javed, Sylvain Kubler, Avleen Malhi, Antti Nurminen, Jérémy Robert, and Kary Främling. bIoTope: Building an IoT Open Innovation Ecosystem for Smart Cities. *IEEE Access*, 8:224318–224342, 2020. Conference Name: IEEE Access. doi:10.1109/ACCESS.2020.3041326.

[25] Kubernetes Project. Controllers - Kubernetes Documentation, February 2021. URL: https://kubernetes.io/docs/concepts/architecture/controller/.

[26] Kubernetes Project. Kubernetes: Production-Grade Container Orchestration, 2021. URL: https://kubernetes.io/.

[27] Kustomize Project. Kustomize - Kubernetes native configuration management, 2021. URL: https://kustomize.io/.

[28] Canonical Ltd. JAAS - Juju as a Service | Juju, 2021. URL: https://jaas.ai/.

[29] B. D. Martino, A. Esposito, and G. Cretella. Semantic Representation of Cloud Patterns and Services with Automated Reasoning to Support Cloud Application Portability. *IEEE Transactions on Cloud Computing*, 5(4):765–779, October 2017. doi:10.1109/TCC.2015.2433259.

[30] Matt Rutkowski, Chris Lauwers, Claude Noshpitz, and Calin Curescu. TOSCA Simple Profile in YAML Version 1.3, February 2020.

[31] Giuseppe Muntoni, Jacopo Soldani, and Antonio Brogi. Mining the Architecture of Microservice-Based Applications from their Kubernetes Deployment. In Christian Zirpins, Iraklis Paraskakis, Vasilios Andrikopoulos, Nane Kratzke, Claus Pahl, Nabil El Ioini, Andreas S. Andreou, George Feuerlicht, Winfried Lamersdorf, Guadalupe Ortiz, Willem-Jan Van den Heuvel, Jacopo Soldani, Massimo Villari, Giuliano Casale, and Pierluigi Plebani, editors, *Advances in Service-Oriented and Cloud Computing*, Communications in Computer and Information Science, pages 103–115, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-71906-7_9.

[32] A. Palesandro, M. Lacoste, N. Bennani, C. Ghedira-Guegan, and D. Bourge. Mantus: Putting Aspects to Work for Flexible Multi-Cloud Deployment. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 656–663, June 2017. doi:10.1109/CLOUD.2017.88.

[33] Kubernetes Project. Using Admission Controllers - Kubernetes Documentation, February 2021. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/.

[34] J. Santos, T. Vanhove, M. Sebrechts, T. Dupont, W. Kerckhove, B. Braem, Gregory Van Seghbroeck, T. Wauters, P. Leroux, S. Latre, B. Volckaert, and Filip De Turck. City of Things: Enabling Resource Provisioning in Smart Cities. *IEEE Communications Magazine*, 56(7):177–183, July 2018. doi:10.1109/MCOM.2018.1701322.

[35] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM. doi:10.1145/2465351.2465386.

[36] Merlijn Sebrechts. IBCNServices/kubernetes-relationships-results, 2021. URL: https://github.com/IBCNServices/kubernetes-relationships-results.

[37] Merlijn Sebrechts, Sander Borny, and Lennart Onghena. IBCNServices/orcon, February 2021. original-date: 2020-03-04T21:36:45Z. URL: https://github.com/IBCNServices/orcon.

[38] Merlijn Sebrechts, Sander Borny, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Model-driven deployment and management of workflows on analytics frameworks. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2819–2826, December 2016. doi:10.1109/BigData.2016.7840930.

[39] Merlijn Sebrechts, Cory Johns, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Beyond Generic Lifecycles: Reusable Modeling of Custom-Fit Management Workflows for Cloud Applications. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 326–333, July 2018. doi:10.1109/CLOUD.2018.00048.

[40] Merlijn Sebrechts, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert, and Filip De Turck. Orchestrator conversation: Distributed management of cloud applications. *International Journal of Network Management*, 28(6):e2036, 2018. doi:10.1002/nem.2036.

[41] Fikret Sivrikaya, Nizar Ben-Sassi, Xuan-Thuy Dang, Orhan Can Görür, and Christian Kuster. Internet of Smart City Objects: A Distributed Framework for Service Discovery and Composition. *IEEE Access*, 7:14434–14454, 2019. doi:10.1109/ACCESS.2019.2893340.

[42] Luis M. Vaquero, Felix Cuadrado, Yehia Elkhatib, Jorge Bernal-Bernabe, Satish N. Srirama, and Mohamed Faten Zhani. Research challenges in nextgen service orchestration. *Future Generation Computer Systems*, 90:20–38, January 2019. doi:10.1016/j.future.2018.07.039.

[43] VMWare, Inc. What is ESXI? | Bare Metal Hypervisor | ESX, 2021. URL: https://www.vmware.com/products/esxi-and-esx.html.

[44] Sebastian Wagner, Uwe Breitenbücher, Oliver Kopp, Andreas Weiß, and Frank Leymann. Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans. In *Cloud Computing and Services Science*. Springer, Cham, April 2016. doi:10.1007/978-3-319-62594-2_12.

[45] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, and K. Saatkamp. Standards-Based Function Shipping - How to Use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments. In *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*, pages 50–60, October 2017. doi:10.1109/EDOC.2017.16.

[46] Polona Štefanič, Matej Cigale, Andrew C. Jones, Louise Knight, Ian Taylor, Cristiana Istrate, George Suciu, Alexandre Ulisses, Vlado Stankovski, Salman Taherizadeh, Guadalupe Flores Salado, Spiros Koulouzis, Paul Martin, and Zhiming Zhao. SWITCH workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications. *Future Generation Computer Systems*, 99:197–212, October 2019. doi:10.1016/j.future.2019.04.008.

5

Joint Management of Serverless Stream Processing Pipelines Crossing Organizational Boundaries

This chapter answers the research question "How can multiple independent parties collaboratively create serverless streaming pipelines?". Serverless pipelines are a unique challenge for service orchestration because the platform is expected to provide both control-plane and data-plane functionality. This is in contrast to orcon, for example, which resides purely in the control plane. Although orcon can configure communication channels, it does not handle the actual communication itself. In serverless platforms, however, the complexity of communication between workloads is moved into the platform. This chapter presents "Plumber" to address this challenge. It is a framework for building and running serverless stream processing pipelines that cross organizational borders. It has a specific focus on enabling collaborative creation of such pipelines, a focus that is lacking in current state of the art. It goes beyond that focus, however, to deliver a user-friendly UI and advanced features such as atomic upgrades, automatic scaling and seamless roll-back to previous versions. Like the orcon solution from Chapter 4, it is also a Kubernetes-native framework ensuring high compatibility with the existing ecosystem.

⋆ ⋆ ⋆

M. Sebrechts, B. Verstraete, S. Borny, F. De Turck, and B. Volckaert

Submitted for review, July 2022.

Abstract Collaboration across organizational boundaries, whether it is between teams, organizations or companies, is becoming increasingly important. Both industry 4.0 and smart cities require collaboration and sharing of data to succeed. As a result, Gartner predicts that by 2023, organizations which have the technology and processes to enable interenterprise data sharing will outperform those that do not. The current state of the art in stream processing platforms, however, does not offer cohesive facilities for such collaboration. Existing technologies lack methods for collaboratively modelling and building data processing topologies and lack hands-off change management techniques.

This research proposes a method to allow for asynchronous collaboration in both the modelling and change management of serverless stream processing topologies. The introduced concepts are illustrated through the development of Plumber: an open source Kubernetes native framework that allows for the iterative development and management of processing topologies in a serverless manner. A functional evaluation shows the platform makes it possible to collaboratively create stream processing pipelines that cross organizational borders. A performance evaluation shows the framework can update a running cross-domain analytics pipeline in 12 seconds without any data loss or duplication.

5.1 Introduction

Every day, organizations have to deal with an increasing amount of streaming data [18, 2, 47]. As the volume of data increases, so does the need to distill this data into increasing amounts of beneficial information. Stream processing frameworks have been able to fill the niche of dealing with these increasingly real-time processing needs. The fourth industrial revolution adds an additional challenge to this need, however. Lepore et al. show that collaboration between teams, organizations and companies is imperative for introducing industry 4.0 technologies [29]. This creates a high need for interconnecting systems and sharing data [8]. As more and more of these systems get interconnected, data streams start to cross organizational boundaries are jointly managed by multiple independent parties. This need is not limited to the factory floor, however. Gartner predicts that by 2023, organizations which have the technology and processes to enable interenterprise data sharing will outperform those that do not [15]. As technology becomes more and more ingrained into society, so does the need for IT systems which can handle the realities of collaboration without complete trust. Smart cities, for example, require collaboration between government and multiple industry partners, each with their independent goals. Solving the problems of modern cities requires these parties to work together up to the data stream level. Finally, even within a single company, stream processing often requires collaboration. As an example, two roles typically work closely together on these projects: the data scientist and the data engi-

neer. The scientist is interested in gaining valuable insights from data and possesses expertise in data analysis, machine learning, etc. The engineer is in charge of wrangling raw data from various sources and supporting the scientist's study by operating stream processing frameworks and other data infrastructures [7].

Serverless is a paradigm that simplifies operating an application by providing an opinionated abstraction over all layers except the core business logic [4]. Applied to stream processing, users only need to supply their business logic and minimal configuration to form a running topology. Many operational concerns, such as scaling, scheduling, and observability, are taken care of by the serverless platform [16]. Simultaneously, serverless approaches have the potential to reduce resource usage [1, 34], even outside of the cloud in an IoT setting [48, 36]. There is still a technology gap between stream processing and serverless solutions, however, such as in the area of moving state from one processor to the next [23]. Current serverless solutions on Kubernetes such as Knative [27, 25], for example, lack first-class support for the composition of functions into topologies [5]. As a result, changes that touch multiple parts of a processing topology require manual, error-prone work. More recently, serverless workflow tools have popped up to facilitate real-time use cases. A notable example of this is Argo Dataflow [3]. These tools, however, still lack in the form of change management. This blind spot is becoming increasingly challenging as companies release faster and more often using methodologies such as Agile and Design thinking [11]. This increases the need for software changes to happen flawlessly, and to easily roll-back changes after an issue is detected. Nevertheless, the current generation of stream processing frameworks still require manual work to ensure that no messages are lost or duplicated during upgrades. Finally, serverless topologies are often modeled as single entities, with implicit links between topologies. This makes it difficult for multiple parties to collaboratively manage topologies that span organizational boundaries.

This research aims to tackle these issues by developing Plumber [21]: an open source platform for creating and managing serverless stream processing pipelines which cross organizational boundaries. Specifically, Plumber aims to provide the following properties.

- **Cross-organizational topologies**: Allow multiple parties to create and manage different parts of a single serverless function composition.
- **Asynchronous collaboration**: Allow these parties to independently and asynchronously manage their respective parts.
- **Explicit, declarative models**: Provide a declarative API that allows users to explicitly define topologies in a versioned manner, so users have a clear view of currently and previously running pipelines. Users specify a topology in a declarative manner, so that the platform itself is responsible for taking the necessary actions to deploy the pipeline and ensure it is running up to specification.
- **Serverless, zero-touch change management**: The platform automatically manages and scales pipelines, and takes care of the boilerplate configuration, allowing users to focus

on the processing logic. The platform supports at-least-once processing semantics and ensures these are met at all times, including during upgrades.

- **Kubernetes Native**: As an extension to cloud native; provide deep integration into the Kubernetes ecosystem so compatibility with existing tools is ensured, and so that the platform can run on any Kubernetes-certified cluster.

This article starts by presenting a more in-depth look into the state of the art in Section 5.2. The motivating use-case is presented in Section 5.3. The design and architecture of the proposed framework is discussed in Section 5.4. Section 5.5 takes a deep dive into how the platform processes changes to running topologies. The functionality and performance of the resulting framework are evaluated in Section 5.6 and Section 5.7 respectively. Section 5.8 concludes the work and Section 5.9 gives a glimpse into future work in this space.

5.2 Related Work

Creating, deploying and managing chains of serverless functions connected by dependencies is extensively studied in related work, although multiple terms are used to refer to broadly similar concepts. Serverless function compositions [24], serverless topologies [49], serverless dependencies [30], serverless function chains [13], serverless workflows [37], and serverless choreographies [38] are all used in related work to refer to graphs of serverless functions.

A number of approaches exist for tracing application dependencies of distributed applications at runtime [30, 41]. *Lowgo*, for example, records dependencies of serverless function chains across clouds [30]. This approach is very useful for mapping the topology of existing serverless applications, and could be extended for topologies crossing organizational boundaries. The downside of this approach, however, is that it is focused on runtime observability instead of design-time and operational support. Specifically, during design and modification of topologies, developers still need to manually manage dependencies.

A common issue when moving computation from the cloud towards the edge is the heterogeneity of underlying infrastructure and resources [32]. One approach aiming to solve this issue is symbIoTe [43, 50], creating unified search and control for disparate IoT resources across management domains. Chen et al. introduce FogSEA [9], which uses a fully decentralized service composition model to deploy and connect services at runtime. Risco et al. introduce a platform for serverless workflows running in the Cloud Continuum [37]. While these approaches tackle the issue of managing applications running on *heterogeneous infrastructure*, they do not address the issue of managing applications created by a *heterogeneous development team* that crosses organizational boundaries.

One approach for enabling composition of serverless functions is by implementing composition via reflection in the serverless runtime, called *composition-as-function* by Baldini et al. [5]. An advantage of this solution is that it enables nested compositions since a composition of functions

is itself also a function. This feature, however, introduces a challenge for compositions which cross organizational boundaries. While it is clear who owns and has access to individual *child* functions, this is not straightforward for the composition or *parent* function itself.

A second approach for serverless function composition can be called *composition-as-topology*: a two-level system in which individual functions are composed into a *topology* or a *workflow*. A function and a topology are functionally very distinct and are handled very differently in such a platform. As a result, this approach does not enable nested compositions and thus does not suffer from the same issues as composition-as-function for cross-domain compositions. Although this approach is widely used in related work [40, 42, 10, 14, 49, 44, 39, 31, 38], no existing approach addresses the challenge of cross-domain compositions.

Datta et al. aim to improve the security of serverless applications through auditing of network traffic using a proxy [12]. Although this approach allows more secure re-use of third-party code, each serverless topology is still contained in a single management domain, resulting in an "all or nothing" permission model for developers modifying these topologies.

5.3 Motivating use-case

To further illustrate the challenges addressed in this research and the requirements of the proposed framework, we present a motivating use-case of a Data Scientist and Data Engineer, working together on a predictive maintenance and event-based actuation pipeline.

- The **Data Scientist** is responsible for creating and maintaining algorithms that analyze a stream of IoT data in order to decide which actuations to take, and to predict when maintenance should be performed.
- The **Data Engineer** is in charge of ingesting raw data from various IoT sensors, decoding it, and making it available for analysis. Secondly, the Data Engineer is also responsible for receiving the output of the analysis and moving it to the various platforms which act on this data.

The end result is a stream processing pipeline that ingests raw events from IoT sensors, decodes the data, and splits into two parallel branches, each of which processes a copy of all decoded events.

- The first branch removes all invalid events from the stream and subsequently analyses the events in order to generate actuations.
- The second branch analyses the decoded events to predict when maintenance is required. If maintenance is required, this branch emits an event which is sent to the maintenance department.

Both the Data Scientist and Data Engineer want to independently make changes to this topology. The data engineer wants to update the ingest source and egress destinations, and update the

decoding logic when issues are detected or changes are made in connected systems. The data scientist wants to iterate on the analysis logic to improve it or to meet new business needs.

As a result, there is a need for a stream processing framework which allows two independent parties to create and update different parts of a single stream processing topology. The solution should also take into account a number of secondary requirements as stated in Section 5.1.

- Declarative definition of pipelines and seamless rollback of the pipeline to previous versions.
- Horizontal autoscaling out of the box.
- Atomic no-touch upgrades from the standpoint of the stream such that there is a single upgrade point after which all new messages are processed by the upgraded topology.
- At-least-once processing semantics at all times.
- Using the Kubernetes API for interaction with users.

5.4 Architecture

This section outlines the overall architecture of Plumber [21]. This takes into account the requirements originating from the motivating use-case in Section 5.3.

Plumber's architecture is divided into four layers, discussed in-order in the remainder of this section.

- The **domain model** describes the abstract concepts which constitute a Plumber serverless stream processing topology.
- The **data plane** contains the components which process events and run the serverless functions.
- The **management interface** contains the custom Kubernetes resources that users interact with to build and manage topologies collaboratively.
- The **control plane** contains the components that drive the data plane towards the user-specified desired state.

5.4.1 Domain model of composition as topology

As explained in section 5.2, one way to support compositions of serverless functions is using the composition-as-function paradigm in which each composition is also a function. Although this approach allows creating new serverless abstractions, and thus promotes reuse, it introduces some problems when serverless compositions cross organizational boundaries. The problems stem from the property that different parts of the composition have different ownership. As such, there is no single party which completely owns the composition and no single party which can reuse the

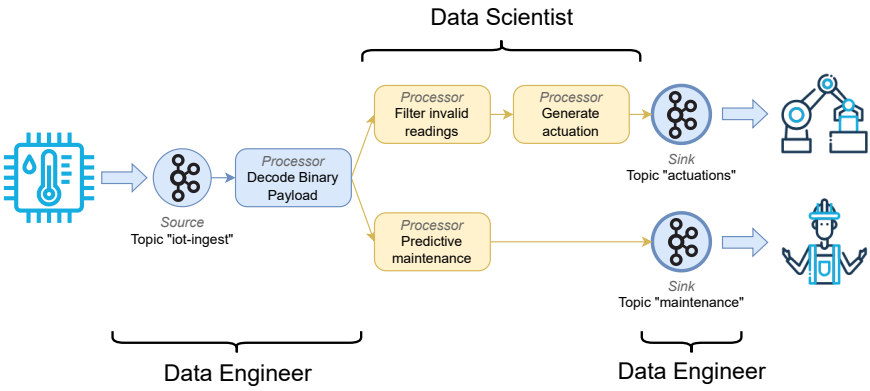

Figure 5.1: A Plumber stream processing topology crossing organizational boundaries. The Data Engineer and Data Scientist both manage different parts of the same topology, requiring collaboration in order to get the desired result.

composition. For this reason, our solution uses the two-level composition-as-topology paradigm. Functions are composed by connecting them to each other into a Direct Acyclic Graph (DAG). This approach is common in stream processing frameworks such as Apache Storm, Spring Cloud Stream and Kafka streams.

In Plumber, a topology consists of four distinct types of nodes.

- **Sources** are declarative abstractions over external event emitting systems. The source acts as an event ingress into topologies. Each source has one *input* event stream.
- **Processors** are units of code that takes events as input, run a serverless function on them, and produce output events. Each processor has one *input* and one *output* event stream.
- **Sinks** are declarative definitions of egress systems. Each sink has one *output* event stream.

Users create directed acyclical processing topologies by creating such nodes and connecting each input of a node to the output of a previous node using the following rules.

- Each output must be connected to the input of *at least* one other node.
- Each input must be connected to the output of *exactly* one other node.
- The output of a source cannot be directly connected to the input of a sink.
- The topology must be a DAG.

Figure 5.1 shows how these domain objects can be used to create the topology of the motivating use-case explained in Section 5.3. The Data Engineer is responsible for one source, one processor and two sinks. The Data Scientist is responsible for three processors. Note that it is possible for

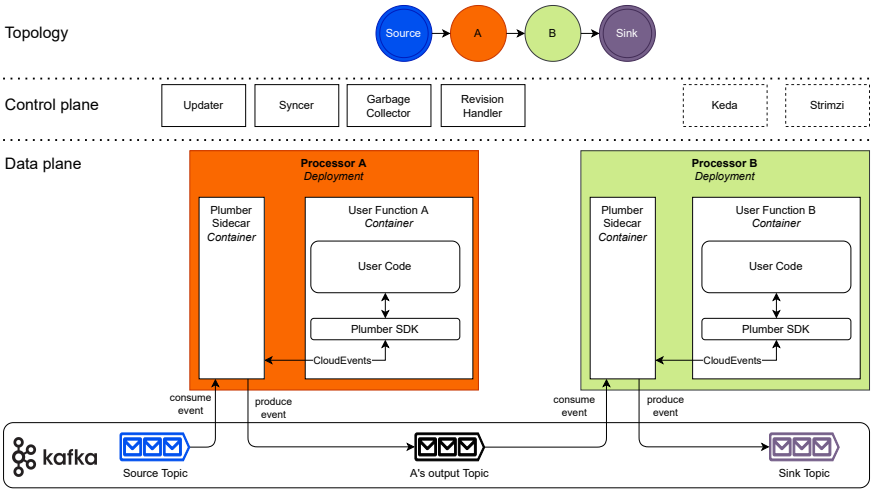

Figure 5.2: High-level Plumber architecture showing with color-coding how a topology of domain objects corresponds with data-plane components.

a node to send its result to two other nodes, even though each node has only one output stream. This is because each output stream can be connected to multiple input streams.

5.4.2 Data Plane

The data plane of Plumber consists of a number of containers running in Kubernetes communicating using Apache Kafka topics. As shown in Figure 5.2, each Processor is a Kubernetes Deployment consisting of two containers: the *sidecar* and the *user function*.

The **sidecar** can be seen as a proxy between the user function and the rest of the data plane. It embeds all necessary logic to facilitate data plane communication. The sidecar further canonicalizes events into a common format, *CloudEvents*, before sending them to the user function. The sidecar forwards the transformed event based on the user function response, or drops the event if the user function decides so. This last property makes it possible for a function to act as a filter, removing invalid or otherwise unneeded events.

The **user function** embeds the user supplied code. Although it is possible for user code to manually respond to and parse CloudEvents, Plumber also supplies a Python SDK to reduce the amount of boilerplate code needed. This SDK handles sidecar interaction, provides logging facilities, and has a minimal interface for the receiving of events and taking actions such as dropping an event or forwarding the transformed data.

Apache Kafka is used for communication between components. Each Source and Sink is a Kafka topic, and Processors communicate by publishing to- and consuming from Kafka topics. Plumber uses Kafka in at-least-once configuration. This configuration is ensured by the Plumber sidecars,

which make sure to configure Kafka clients and handle Kafka communications in the correct manner to get these guarantees. This limits the scope of user error to affect the message guarantees.

5.4.3 Management Interface

The management interface offers three custom resources for users to interact with:

- **TopologyPart** represents a part of a topology which is managed by a single team. It is a collection of the desired state of the nodes and edges in that part of the topology. The edges are defined by referencing other nodes by name, to either take input from or output towards. These references connect the corresponding event streams of those nodes following the rules explained in Section 5.4.1.
- **TopologyPartRevision** is an immutable snapshot of a TopologyPart; a new snapshot is automatically created upon each change to a TopologyPart.
- **Topology** represents a complete topology created by combining TopologyPartRevisions. It is a collection of references to TopologyPartRevisions which together form the desired state of a single topology.

Together, they provide declarative and collaborative (change-)management of topologies. Data engineers and scientists can independently build and change the parts they are responsible for. Once all appropriate changes are made, one of them can change the Topology to encapsulate the new, combined desired state as expressed in the TopologyPartRevisions.

5.4.4 Control Plane

The control plane as shown in Figure 5.3 is modeled as a set of collaborating Kubernetes controllers. The first set of controllers are custom developed for Plumber as part of this research:

- The **Updater** is responsible for correctly orchestrating transitions between versions of Topologies.
- The **Syncer** translates the desired state of the active version of the Topology into its data plane components.
- The **Garbage Collector** cleans up data plane components of Topology versions that are not required anymore.

The second set of operators are existing third party open source projects which are used by Plumber to perform more generic tasks:

- **KEDA** provides auto-scaling capabilities for event-driven workloads in Kubernetes [26]. KEDA allows Plumber to scale Processors reactively based on Kafka topic depth.
- **Strimzi** brings management of Kafka into Kubernetes [35], allowing for declarative management of clusters, topics, and configuration.

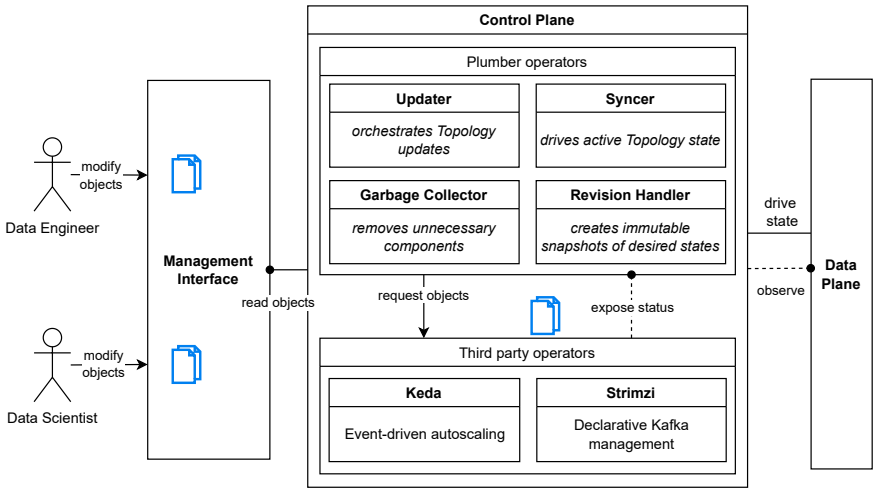

Figure 5.3: Interactions within the control plane. All parties interact through the management interface by modifying Kubernetes objects which represent the desired state of their part of the topology.

5.5 Change Management Flows

The control plane must consider common user interaction scenarios concerning change management. Therefore, this section dives deeper into the two most important flows: handling updates to a Topology and the subsequent garbage collection.

5.5.1 Update Flow

When users make a semantic change to a Topology and another version of the Topology is already running, the control plane takes care of meeting the following properties:

- **At-least-once guarantees**: Should remain respected during updates; all events ingested from a Source by a Topology version should be fully processed by that version.
- **No duplicate processing**: Zero duplicated events should be processed as a result of the update mechanism.
- **Limit processing downtime**: The period in which no version actively processes from a Source should be minimized.
- **Atomicity**: Upgrades should be atomic, such that no two versions process events from a Source concurrently.
- **Convergence**: Convergence towards the desired state should be achievable in all cases, even when the controllers themselves crash momentarily.

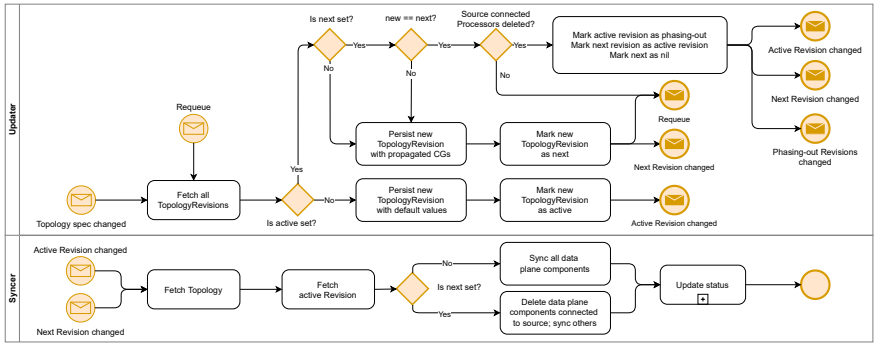

Figure 5.4: Change management Flows of the Updater and Syncer.

The concept of Revisioning is applied to Topologies in the form of *TopologyRevisions* to enable these properties. These Revisions embed the full combined desired state of the Topology alongside configurations that should carry over between Revisions. The TopologyRevisions have three distinct states; these states are persisted on the respective Topology object themselves. Transitions of these states happen strictly through the Updater. The following states are defined:

- **Active**: The Revision that should be actively Processing new events from the Sources.
- **Next**: The Revision that should take over Source processing from the Active revision.
- **Phasing-Out**: A Revision that was previously Active, but may still have outstanding events in its internal topics to be processed.

Upon detecting a semantic change to a Topology and its composed desired state, the Updater creates a TopologyRevision as shown in Figure 5.4. In the case of an update, an *active* Revision is already set. The Updater ensures that all necessary information for processing continuation, i.e., the correct starting offsets and consumer groups, are propagated to the new Revision. Subsequently, the Updater marks the newly created Revision as the *next* Revision. The Syncer is notified upon this occurrence and deletes the Source-connected Processors of the *active* Revision from the data plane. When the Updater notices that the Syncer has completed this work, it marks the *next* Revision as *active* and the *active* Revision as *phasing-out* in one atomic operation. Of course, the Syncer then brings the new *active* Revision to the data plane immediately.

The initial creation of a Topology is a special case of the Update flow, in which no *active* Revision is present yet. Upon noticing this, the Updater immediately creates a TopologyRevision with default consumer group settings and immediately marks it as *active*. Subsequently, the Syncer continues to translate the *active* Revision towards the data plane.

The Syncer passes in the required dataflow configurations to the sidecar, e.g., the input and output topics to use, consumer groups, etc., based on the links made by users and information that spans Revisions added by the Updater.

With this high-level approach, the sub-requirements are all tackled.

- **At-least-once guarantees**: Only the *phasing-out* Revisions' Source-connected Processors are deleted in the upgrade flow. Thus, the Revision can still process all in-flight events.
- **No duplicate processing**: This is ensured by having unique Processor output topics per Revision, i.e., topics not linked to Sinks or Sources. Secondly, Source-connected Processors' consumer groups are propagated between the *active* and the *next* Revision such that after decoupling, the processing continues right where the previous Revision stopped.
- **Limit processing downtime**: The new Revision is brought to the data plane as soon as the previous Revision is decoupled from its Sources.
- **Atomicity**: Grouped state transitions happen through atomic operations.
- **Convergence**: All implemented operations are idempotent and result in changes that are verifiable.

5.5.2 Garbage Collection

The remaining non-Source-connected Processors of *phasing-out* Revisions will eventually process all outstanding events in their input topics. The Processors and their underlying data plane components can be deleted upon verifying this.

When a Revision is marked as *phasing-out*, only the output topic of the deleted Source-connected Processors remains, and no new events will be produced into this topic. As soon as the immediate downstream Processors have drained all of the contained events within the topic, the downstream Processors can be deleted. Again, the Processors that are next in the topological ordering will not have any new events flow into their input topics.

Using this property, the Garbage Collector can perform incremental garbage collection. It starts by determining which Processors are still running and have no running predecessors. The Garbage Collector then checks each of these Processors for completion. If one of the Processors is complete, its deployment is deleted, and the Garbage Collector adds the deleted Processors' immediate downstream neighbors to the list of Processors to check. If the Garbage Collector can verify that all Processors are deleted after a complete run of this procedure, the Garbage Collector can delete the Revision from the *phasing-out* list of the Topology. Otherwise, the Garbage Collection algorithm is performed again at a later time.

5.6 Functional Evaluation

This section evaluates the overall function of the framework to support cross-domain workflows and the function of the KEDA autoscaler.

5.6.1 Cross-domain collaboration

To evaluate whether Plumber indeed enabled cross-domain collaboration, this section implements the motivating use-case presented in Section 5.3 using Plumber. Figure 5.1 is a visual representation of the resulting topology created using the domain objects of Plumber. The code for this implementation can be found on GitHub [19].

The Data Engineer creates the first part of the topology: a TopologyPart that configures one source, one processor that decodes it, and two sinks going to two separate Kafka clusters.

```
apiVersion: plumber.ugent.be/v1alpha1
kind: TopologyPart
metadata:
  name: use-case-engineer
spec:
  sources:
    iot-ingress:
      bootstrap: kafka-cluster-iot:9092
      topic: iot-ingest
  sinks:
    actuation-sink:
      bootstrap: kafka-cluster-iot:9092
      topic: actuations
    maintenance-sink:
      bootstrap: kafka-cluster-maintenance:9092
      topic: maintenance
  processors:
    decoder:
      inputFrom: iot-ingress
      image: "decoder-function:v0.0.1"
      maxScale: 8
```

The Data Scientist creates the second part of the topology: a TopologyPart with three processors, two of which ingest from the decoder function specified by the Data Engineer. The end result is sent to two sinks.

```
apiVersion: plumber.ugent.be/v1alpha1
kind: TopologyPart
metadata:
  name: use-case-scientist
spec:
  processors:
    filter:
      inputFrom: decoder
      image: "filter-function:v0.0.1"
      maxScale: 8
    actuation-generator:
      inputFrom: filter
      image: "actuation-generator-function:v0.0.1"
      sinkBindings: actuation-sink
      maxScale: 24
    maintenance-predictor:
      inputFrom: decoder
      image: "predictive-maintenance:v0.0.1"
      sinkBindings: maintenance-sink
      maxScale: 12
```

Behind the scenes, Plumber creates two TopologyPartRevisions to capture a snapshot of the TopologyParts. The Data Scientist then creates a Topology which uses these revisions to create the full topology.

```
apiVersion: plumber.ugent.be/v1alpha1
kind: Topology
metadata:
  name: use-case-topology
spec:
  parts:
  - name: use-case-engineer
    revision: 1
  - name: use-case-scientist
    revision: 1
  defaultScale: 5
```

Once this topology is submitted, Plumber creates the required data-plane components in order to set up the stream processing pipeline. Once it is submitted, both the Data Scientist and Data Engineer can change their respective TopologyParts without impacting the running pipeline. Only once the Topology specifies a new revision for one of its parts, will the data plane be updated.

This example confirms Plumber can be used to manage stream processing pipelines crossing organizational boundaries. Each individual party creates a TopologyPart, which are joined together in a Topology.

5.6.2 KEDA autoscaling

The current version of Plumber allows all Processor deployments to scale up and down transparently and scale-to-zero if no new events are outstanding. This still requires some configuration from the user, however: they still need to hint at a maximum scale for each of their Processors through the management interface if they would require a higher scaling bound than the default

value. As such, users need to manually override the scale of the Topology or TopologyPart upon realization that the set maximum scale is insufficient for the use case. Ideally, the number of partitions is scaled up transparently if the number of Processors is maxed out. Furthermore, there are three significant issues with the current approach that hinder the practicality of the autoscaling, which are all caused by the reactiveness of KEDA.

The first issue is that scaling operations might have side-effects that hinder continuous processing. When a processor gets scaled, the sidecars and the Kafka consumers inside of them scale accordingly. This triggers Kafka to initiate a rebalancing process. During this process, Kafka reassigns partitions to the existing consumers within the group and offers a mechanism for communicating the partition's last committed offset to the responsible consumer. During this rebalancing process, consumers are temporarily paused, possibly increasing the backlog of events on the topic in Kafka. This increasing backlog can trigger KEDA to scale again, causing another rebalancing process. This mechanism can cause KEDA to continually over-scale, adding more and more instances to handle the load, which each trigger another rebalancing procedure. This effect can be partially alleviated by setting an upscaling stabilization window which pauses KEDA actions for a predefined time after a scaling action. Increasing this window, however, reduces reactiveness of the overall scaling which is not ideal when handling unpredictable load patterns.

The second issue is that it is infeasible for Plumber to automatically tweak the KEDA scaling settings such as the queue depth to workload instance ratio, because this requires knowledge of the processing latencies of each individual Processor in the Topology. Although KEDA has knowledge about the input queues of a processor, there is no way for it to take into account the output of a processor in order to calculate processing latency.

Finally, KEDA is not topology-aware. In scaling decisions, KEDA only takes into account the backlog of the respective processor. As a result, if the event rate greatly increases in the Source of a topology, KEDA will only scale the first processor accordingly. Scaling of a subsequent processor will only happen once enough events are handled by the previous processor to create a backlog for the next processor. As a result, the scaling of the entire topology happens in a slow cascade from upstream to downstream Processors. The overall scaling reactiveness of a topology to increased input load is thus slow.

5.7 Performance Evaluation

5.7.1 Benchmark setup

These benchmarks are executed on a single server with 24 GiB RAM and two Intel Xeon E5645 @ 2.40 GHz processors running Ubuntu 20.04.2 LTS. Kubernetes version 1.20 uses Docker version 20.10.8 and is installed using Kind 0.10.0. All ram and CPU of the host machine is available to the Kubernetes cluster. A single Kafka broker running version 2.7.0 uses three Zookeeper nodes and, are configured with ephemeral storage configurations.

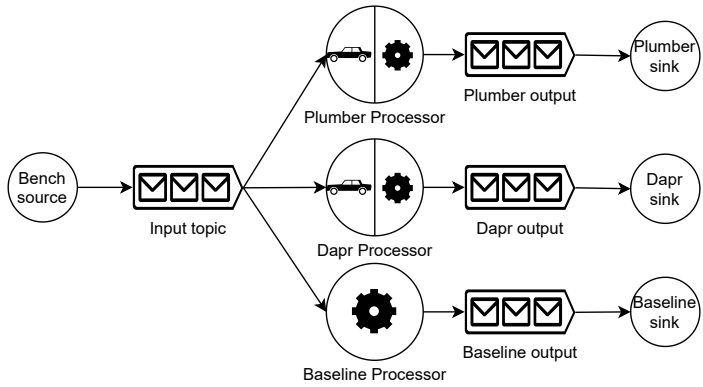

Figure 5.5: Sidecar latency benchmark setup.

5.7.2 Sidecar Latency

Although Plumber's sidecar approach increases extensibility, this includes an additional component in the data path, which adds latency. This benchmark aims to understand the performance penalty of this approach by comparing the end-to-end latency of three pipelines with identical processing logic as shown in Figure 5.5.

- **Baseline**: a single user function container that directly consumes and produces to Kafka. This setup serves as a baseline for the minimum achievable latency.
- **Plumber**: a deployment consisting of the Plumber sidecar and a user function using the Plumber SDK.
- **Dapr**: a deployment consisting of a Dapr sidecar and a user function. This setup has been included to compare the Plumber sidecar with a more mature sidecar implementation.

The Golang implemented bench source [20] produces events containing a begin timestamp with nanosecond precision into the input topic. Processors read these from a shared input topic and forward them to an output topic specific to each setup. The Golang sinks add an end timestamp with the same precision and persist these numbers to disk. The delta between the begin and end timestamp is the measured end-to-end latency. To ensure the test isolates sidecar interaction latency, the source produces at a rate of around 10 event per second, which ensures that each setup has ample time to process the event. A total of 10 000 events are processed to provide a large sample size. Furthermore, each of the setups has identical Kafka consumer and producer configurations. Lastly, each of the Processors has exactly one replica running, and each of the topics has one replica and partition.

Figure 5.6 visualizes the benchmark results. The median added latency of the Plumber sidecar is 0.4 ms. The spreads of the sidecar implementations are more significant than the baseline implementation. This difference can be attributed to the use of an extra protocol which comes with

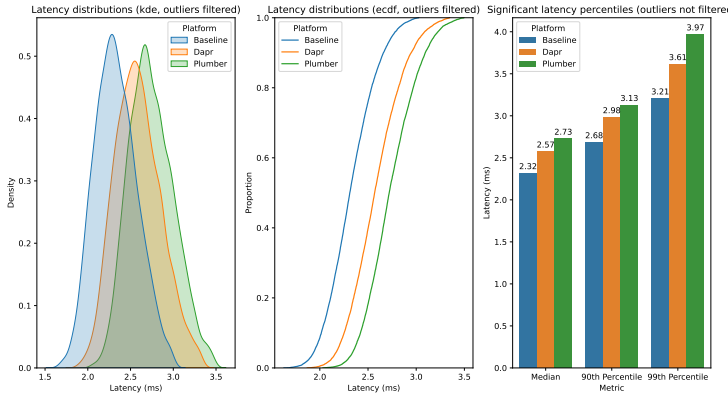

Figure 5.6: The Plumber sidecar adds about 0.4 ms of latency compared to the baseline. This is slightly larger than the Dapr sidecar.

its own variance, causing the latency gap between the sidecar implementations and the baseline implementation to widen as higher percentiles are reached, ultimately causing a 0.8 ms delta between Plumber and the baseline at the 99th percentile. Each of the setups show heavy outliers at the 99.5th percentile, which were filtered from the two density visualizations using the Interquartile Range (IQR) method.

A first observation from the results is that, barring heavy outliers, the results for Plumber are acceptable. The first reason being that Plumber, nor its intended use cases, are focused on minimal latency operations; the solution is tailored to be extensible, and general purpose for stateless operations. The sidecar architecture is important in facilitating this extensibility, as changes to code that impact the runtime are limited to the single sidecar implementation. Secondly, cloud-based serverless solutions such as AWS Lambda, and Azure Functions show invocation latencies ranging from milliseconds to seconds. The right tail, being seconds, is not relevant as this is most likely due to cold-start overhead, a similar overhead is expected upon a scale from zero when using Plumber [28]. Thus, during normal operations, the invocation latency of a cloud serverless function is in the milliseconds and the median Plumber added sidecar latency is a fraction of that. Lastly, the Dapr numbers show promise that the gap between the baseline setup and the Plumber setup can be further narrowed down.

Upon closer inspection of the Dapr sidecar logic, the delta between Plumber and Dapr can be attributed to two implementation details. The first is that Dapr does not enforce the canonicalization of events when reading from a binding, which is similar to a Plumber source. The added conversion code introduces extra allocations; however, this accounts for only a small delta. The primary culprit for the delta is hypothesized to be different HTTP client libraries; Plumber uses the Golang standard libraries' net/http, while Dapr uses a highly optimized open-source library,

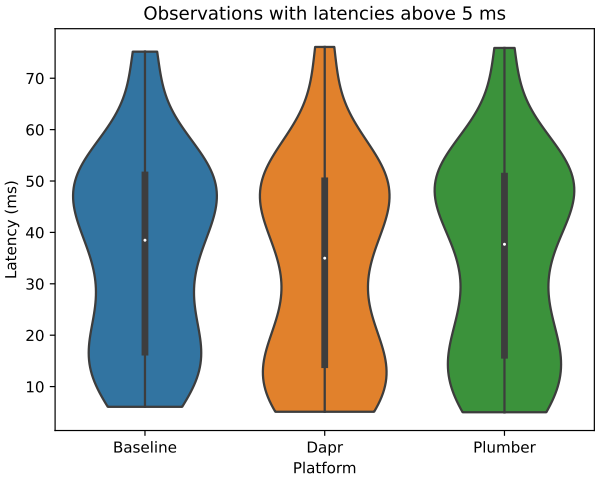

Figure 5.7: Sidecar latency benchmark outliers at a low data rate of 10 events per second.

fasthttp[46]. The self-reported client benchmarks of fasthttp show an up to 10x performance increase over net/http, in part due to its heap avoiding implementation [17]. This data is supported by independent benchmarks conducted by TechEmpower, which compares popular web server framework performances, showing about 3x higher throughput and 3x lower latency is achieved over the net/http package in a simple setup where a JSON-serialized response is returned on a GET request [45].

As mentioned before, about 0.5% of the total events per setup are heavy outliers. These outliers are visualized in Figure 5.7 which zooms in on events with a higher than 5 ms end-to-end latency, the observed 99.9th percentile latency for each platform is about 52 ms. When the data rate is higher, at 800 events per second, the same pattern reoccurs as seen in Figure 5.8. The same general outlier pattern occurs in each of the setups, with a 99.9th percentile latency of about 27 ms for each of the platforms. As the latency is present in each setup, it is not caused by the sidecar mechanism itself but by the Kafka client or server. One likely culprit is the Kafka client batching records before persisting (producing) them, as explained in [33]. With the right timing, subsequent records might arrive in such a manner that a batch is only produced after it is completely full. This would explain why the outlier latency goes down as the data rate goes up, since a batch fills up more quickly in such a scenario. It might also explain why the latency happens in “waves” as the output batch of a sidecar might not fill up completely until a new input batch is ready.

To conclude, the added latency by introducing a sidecar to Plumber is deemed acceptable, as the median latency difference is limited to 0.4 ms compared to a setup without a sidecar. In addition, the extreme tail-end latencies appear to be comparable to a baseline setup, with both having heavy outliers, yet the 99.9th percentile latencies stay acceptable for most use cases.

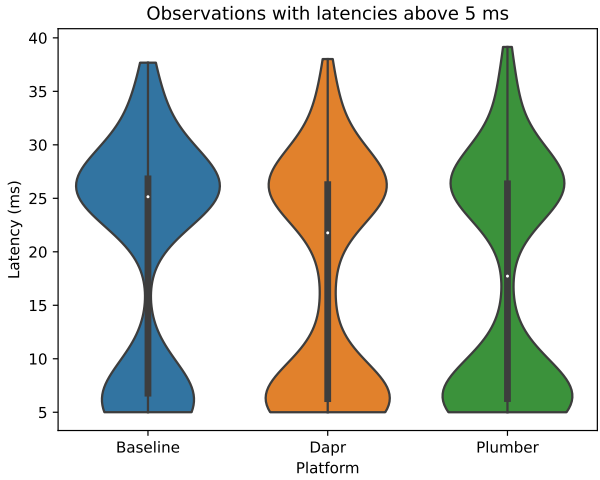

Figure 5.8: Sidecar latency benchmark outliers at a higher data rate of 800 events per second.

5.7.3 Topology Upgrade Speed

This benchmark [20] evaluates the speed and correctness of upgrading a running topology. Specifically, this benchmark checks whether event processing remains at-least-once during an upgrade, and how long an upgrade takes.

This benchmark creates an initial Topology Revision to process a continuous stream of events. After some time, it performs a Topology upgrade. Both revisions of the Topology have a single Processor which simply forwards whatever it reads from a Kafka Source to a Kafka Sink. The benchmark continuously produces messages with a production timestamp and incrementing message ID. Processors wait for a specified period of time, after which they send the message to the next step in the process. Each sink adds a receival timestamp to the message and writes all messages to an append-only file. These message timings and IDs allow for the verification of correctness. First, if there are no gaps in the written message IDs, no events were lost. If there are no duplicate message IDs present, no duplicate processing was performed. Secondly, the operator was instrumented to log the timings and corresponding revision of status transition for bottleneck detection. Each of the following benchmarks was performed twenty times, with the relative gathered timings being averaged out.

Figure 5.9 shows a timeline of important milestones in the Plumber operators after the upgrade is initiated. It further shows the timespans in which the first Revision is processing events, no Revisions are actively processing events, and the second Revision is processing. Note that the overlap time does not imply that the two Revisions are competing for consumption from the Source; it only means that the first Revision is still *phasing-out*.

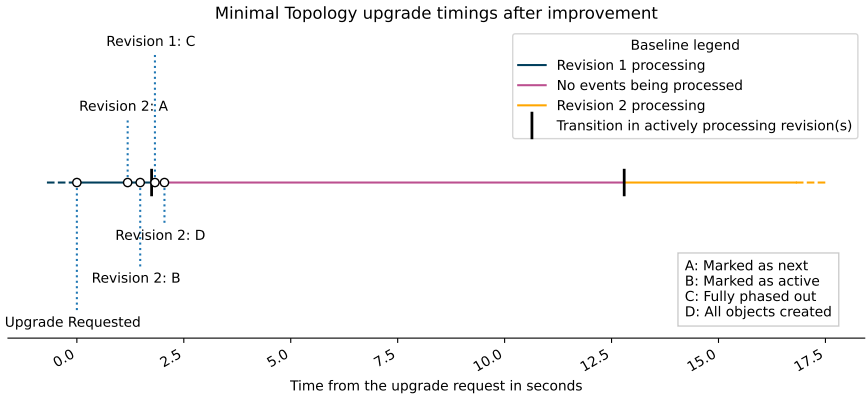

Figure 5.9: Processing of events pauses for about 10 seconds while the containers of the new revision become active.

The collected data of 20 upgrades shows **no events were lost or duplicated** during the upgrade process. A close examination of the timeline reveals a significant period of almost 10 seconds during which processing of events is paused. Specifically, there is a large gap between the second Revision becoming *active* and it processing its first event. This is caused by the time needed for the Processor deployments' underlying pods to be scheduled and run. As such, operators of Plumber topologies should factor in the fact that a pipeline might pause for up to 10 seconds while an upgrade is in progress.

5.7.4 Garbage Collector Benchmark

A third benchmark was conducted with a more elaborate setup, consisting of two Processors in sequence, with the second Processor artificially being slow. With this setup, the Garbage Collector needs to correctly check if all messages have been Processed from the first Processors' output topic by the slow second Processor. Figure 5.10 again shows the relative timings. Notice that there is now a time span where the first and second Revisions are both processing. However, this does not mean both Revisions compete to consume from the Source, as the first Revisions' Source connected Processors are deleted.

Again, no messages were lost, nor were any duplicates observed in 20 runs of an elaborate topology upgrade. Coupled with the previous benchmark results, this experimentally verifies the correctness pertaining to the disallowance of duplicate processing and lost events even when there are unprocessed events for the remaining *phasing-out* Revision. However, as noted before, improvements to the upgrade process should and can be made to mitigate periods where no processing occurs.

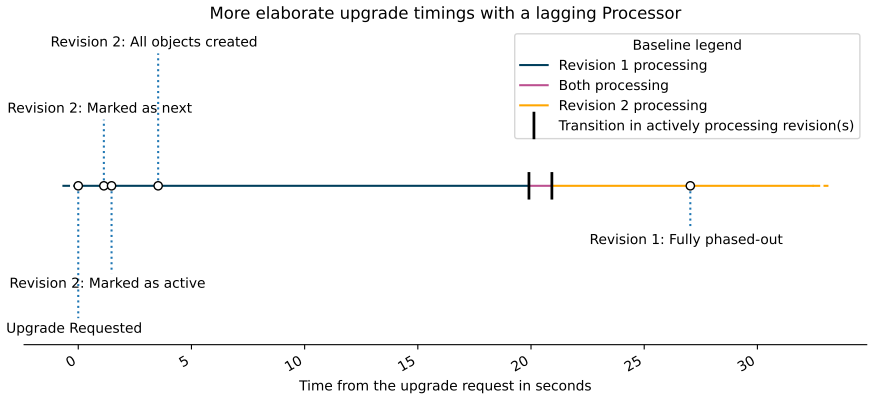

Figure 5.10: A timeline of events during a more elaborate topology upgrade

5.8 Conclusion

Current stream processing solutions lack in two areas: facilitation for streamlined collaboration and support for change management. For these reasons, Plumber, a stateless stream processing framework, was designed and developed in this study to achieve the following key objectives:

- **Streamlined Collaboration**: Users can effectively and collaboratively reuse parts of topologies, compose them, and manage lifecycles of topologies using the proposed management interface. A visual management interface can be implemented as a front-end to the current text-based model to improve the current proposition.

- **Serverless Operations**: Plumber minimizes typical operational work in the building and maintenance of stream processing topologies. Most importantly, this study proposes a generic, no-touch update technique that minimizes operational effort and can be extrapolated towards other (stateless) stream processing technologies.

- **Kubernetes Native**: The implemented framework runs natively in Kubernetes and strictly uses Kubernetes native building blocks. This has two significant implications. Firstly, Plumber integrates with the existing Kubernetes ecosystem through the operator pattern; for example, users may use Continuous Delivery technologies to automate deployments of topologies further. Secondly, the framework can be used on-premises or in the cloud, on any Kubernetes certified distribution of choice.

A thorough evaluation shows the framework handles topology updates in an atomic manner without any message loss, but with a pause of about 10 seconds during which no messages are processed. A comparison of the serverless function overhead shows the sidecar mechanism only adds 0.4ms of additional latency compared to the baseline.

5.9 Future Work

This section described strategies for solving two main shortcomings by optimizing the update flow and improving the overall user experience. Furthermore, a more general proposal to mitigate the shortcomings of KEDA is made.

5.9.1 Optimizing updates

The deployments of the new Revisions' Processors should be created immediately upon an update occurring to enable a faster processing hand-off during the update flow. In addition, the spawned sidecars should be in a standby state until there is a certainty that the previous Revisions' Source-connected Processors were entirely deleted. The sidecars should thus ask the control plane for confirmation to start processing. By immediately creating deployments and having a standby state for the sidecars, the time in which no processing occurs is further mitigated.

5.9.2 Smoother UX

Two main issues hinder a smooth user experience with Plumber. Firstly, to create a function, a user needs to build a Docker container with their code and submit it to a registry accessible by Plumber. Secondly, to create a topology, a user needs to modify and submit Kubernetes objects using a textual representation such as YAML. To improve the UX regarding these issues, a browser-based GUI can be created that serves as a front-end to Plumber and its management interface. The GUI could allow users to write function code in the browser, and have source-to-image conversion to handle the containerization automatically. Secondly, this GUI could allow users to compose and modify Topologies and their Parts in a drag-and-drop manner.

5.9.3 Predictive event-driven autoscaling

The state-of-the-art in event-driven scaling on Kubernetes was found to be insufficient for use in Plumber. The first issue is that KEDA only scales reactively. Future work should investigate how predictive FaaS autoscaling approaches such as from Balla et al. [6] can be integrated into Kubernetes as a whole and Plumber specifically. However, as suggested in Section 5.7, a second issue arises from KEDA's lack of knowledge about the composition's topology. Recent work in the predictive scaling field [22] that takes into account the data-flow dependencies when making scaling discussions could alleviate these issues. The method proposed by the authors requires an offline phase, however, and is not designed for event-driven scaling. Work by Daw et al. is much more promising since it is specifically built for event-driven serverless compositions, and has an optional predictive component [13].

5.9.4 Exactly-once processing

Plumber currently uses exactly-once processing, where each message is guaranteed to be processed at least once. During a fail-over scenario, however, it is possible that a message is processed more than once. For example, this can happen when the sidecar crashes between producing the result of a processed message to the output topic and committing the consumption offset in the input topic to signal the message has been processed. With some modifications to Plumber, Kafka Transactions and Idempotency can be used to ensure this race condition never happens.

Bibliography

[1] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 884–889, New York, NY, USA, August 2017. Association for Computing Machinery. doi:10.1145/3106237.3117767.

[2] Amy Machado. Worldwide Continuous Analytics Software Forecast, 2021–2025. Market Forecast US48047021, International Data Corporation (IDC), July 2021. URL: https://www.idc.com/getdoc.jsp?containerId=US48047021.

[3] Argo Project. Dataflow, July 2022. original-date: 2021-03-02T18:34:05Z. URL: https://github.com/argoproj-labs/argo-dataflow.

[4] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless Computing: Current Trends and Open Problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, Singapore, 2017. URL: https://doi.org/10.1007/978-981-10-5026-8_1.

[5] 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 *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, October 2017. Association for Computing Machinery. doi:10.1145/3133850.3133855.

[6] David Balla, Markosz Maliosz, and Csaba Simon. Towards a Predictable Open Source FaaS. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5, April 2022. ISSN: 2374-9709. doi:10.1109/NOMS54207.2022.9789777.

[7] Christopher Bolard. Data Engineer VS Data Scientist, October 2019. URL: https://towardsdatascience.com/data-engineer-vs-data-scientist-bc8dab5ac124.

[8] Luis M. Camarinha-Matos, Rosanna Fornasiero, and Hamideh Afsarmanesh. Collaborative Networks as a Core Enabler of Industry 4.0. In Luis M. Camarinha-Matos, Hamideh Afsarmanesh, and Rosanna Fornasiero, editors, *Collaboration in a Data-Rich World*, IFIP Advances in Information and Communication Technology, pages 3–17, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-65151-4_1.

[9] Nanxi Chen, Siobhán Clarke, and Shu Chen. Fog-Based Service Enablement Architecture. In *Fog and Fogonomics*, pages 151–177. John Wiley & Sons, Ltd, 2020. URL: https://doi.org/10.1002/9781119501121.ch7.

[10] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. In *2019 IEEE International Conference on Services Computing (SCC)*, July 2019. ISSN: 2474-2473.

[11] M. Ann Garrison Darrin and William S. Devereux. The Agile Manifesto, design thinking and systems engineering. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–5, April 2017. ISSN: 2472-9647. doi:10.1109/SYSCON.2017.7934765.

[12] 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, New York, NY, USA, April 2020. Association for Computing Machinery. doi:10.1145/3366423.3380173.

[13] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 356–370, New York, NY, USA, December 2020. Association for Computing Machinery. doi:10.1145/3423211.3425690.

[14] 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*. IEEE Computer Society Press, May 2022.

[15] Alan D. Duncan. Over 100 Data and Analytics Predictions Through 2025. Technical Report G00744238, Gartner, Inc., March 2021.

[16] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless Applications: Why, When, and How? *IEEE Software*, 38(1):32–39, January 2021. Conference Name: IEEE Software. doi:10.1109/MS.2020.3023302.

[17] fasthttp Authors. HTTP client comparison with nethttp, 2022. URL: https://github.com/valyala/fasthttp#http-client-comparison-with-nethttp.

[18] Forrester Consulting, Inc. The Speed Of Digital Business Demands Streaming Analytics Platforms - Now. Technical report, Forrester Consulting, Inc., October 2019. URL: https://www.cloudera.com/campaign/forrester-speed-of-business-demands-streaming-analytics.html.

[19] Ghent University - imec, IDLab. Plumber use case implementation code, July 2022. original-date: 2022-07-18T16:38:13Z. URL: https://github.com/IBCNServices/plumber.

[20] Ghent University - imec, IDLab. plumber/benchmarks at upgrade-bench · IBCNServices/plumber, February 2022. URL: https://github.com/IBCNServices/plumber/tree/upgrade-bench/benchmarks.

[21] IDLab Ghent University imec. plumber, July 2022. original-date: 2022-07-18T16:38:13Z. URL: https://github.com/IBCNServices/plumber.

[22] Alireza Goli, Nima Mahmoudi, Hamzeh Khazaei, and Omid Ardakanian. A Holistic Machine Learning-Based Autoscaling Approach for Microservice Applications. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER*, pages 190–198. SciTePress Digital Library, February 2021. doi:10.5220/0010407701900198.

[23] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. In *arXiv:1812.03651 [cs]*, Asilomar, California, December 2018. doi:10.48550/arXiv.1812.03651.

[24] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):149:1–149:26, October 2019. doi:10.1145/3360575.

[25] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. Towards Serverless as Commodity: a case of Knative. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, pages 13–18, New York, NY, USA, December 2019. Association for Computing Machinery. doi:10.1145/3366623.3368135.

[26] KEDA Project. KEDA | Kubernetes Event-driven Autoscaling, 2022. URL: https://keda.sh/.

[27] Knative Project. Home - Knative, 2022. URL: https://knative.dev/docs/.

[28] Collin Lee and John Ousterhout. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 149–154, New York, NY, USA, May 2019. Association for Computing Machinery. doi:10.1145/3317550.3321447.

[29] Dominique Lepore, Sabrina Dubbini, Alessandra Micozzi, and Francesca Spigarelli. Knowledge Sharing Opportunities for Industry 4.0 Firms. *Journal of the Knowledge Economy*, 13(1):501–520, March 2022. doi:10.1007/s13132-021-00750-9.

[30] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. Tracing Function Dependencies across Clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260, July 2018. ISSN: 2159-6190. doi:10.1109/CLOUD.2018.00039.

[31] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, DEBS '20, pages 3–14, New York, NY, USA, July 2020. Association for Computing Machinery. doi:10.1145/3401025.3401731.

[32] Pieter-Jan Maenhaut, Bruno Volckaert, Veerle Ongenae, and Filip De Turck. Resource Management in a Containerized Cloud: Status and Challenges. *Journal of Network and Systems Management*, 28(2):197–246, April 2020. doi:10.1007/s10922-019-09504-0.

[33] Anna ovzner and Scott Hendricks. Tail Latency at Scale with Apache Kafka, February 2022. URL: https://www.confluent.io/blog/configure-kafka-to-minimize-latency/.

[34] Ramon Perez, Priscilla Benedetti, Matteo Pergolesi, Jaime Garcia-Reinoso, Aitor Zabala, Pablo Serrano, Mauro Femminella, Gianluca Reali, Kris Steenhaut, and Albert Banchs. Monitoring Platform Evolution Toward Serverless Computing for 5G and Beyond Systems. *IEEE Transactions on Network and Service Management*, 19(2):1489–1504, June 2022. Conference Name: IEEE Transactions on Network and Service Management. doi:10.1109/TNSM.2022.3150586.

[35] Strimzi Project. Using Strimzi (0.27.1), 2022. URL: https://strimzi.io/docs/operators/0.27.1/using.html.

[36] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, January 2021. doi:10.1016/j.future.2020.07.017.

[37] Sebastián Risco, Germán Moltó, Diana M. Naranjo, and Ignacio Blanquer. Serverless Workflows for Containerised Applications in the Cloud Continuum. *Journal of Grid Computing*, 19(3):30, July 2021. doi:10.1007/s10723-021-09570-2.

[38] Sasko Ristov, Dragi Kimovski, and Thomas Fahringer. FaaScinating Resilience for Serverless Function Choreographies in Federated Clouds. *IEEE Transactions on Network and Service Management*, pages 1–1, 2022. Conference Name: IEEE Transactions on Network and Service Management. doi:10.1109/TNSM.2022.3162036.

[39] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. AFCL: An Abstract Function Choreography Language for serverless workflow specification. *Future Generation Computer Systems*, 114:368–382, January 2021. doi:10.1016/j.future.2020.08.012.

[40] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Surveys*, January 2022. Just Accepted. doi:10.1145/3510611.

[41] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204, July 2021. ISSN: 2575-8411. doi:10.1109/ICDCS51616.2021.00027.

[42] Fedor Smirnov, Behnaz Pourmohseni, and Thomas Fahringer. Apollo: Modular and Distributed Runtime System for Serverless Function Compositions on Cloud, Edge, and IoT Resources. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, pages 5–8, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3452413.3464793.

[43] Sergios Soursos, Ivana Podnar Žarko, Patrick Zwickl, Ivan Gojmerac, Giuseppe Bianchi, and Gino Carrozzo. Towards the cross-domain interoperability of IoT platforms. In *2016 European Conference on Networks and Communications (EuCNC)*, pages 398–402, June 2016. doi:10.1109/EuCNC.2016.7561070.

[44] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 311–327, New York, NY, USA, October 2020. Association for Computing Machinery. doi:10.1145/3419111.3421306.

[45] TechEmpower. Golang JSON Serialization Benchmark, February 2021. URL: https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=json&l=zijocf-sf.

[46] Aliaksandr Valialkin. fasthttp, July 2022. original-date: 2015-10-18T22:19:57Z. URL: https://github.com/valyala/fasthttp.

[47] W. Roy Schulte, Pieter den Hamer, and Ehtisham Zaidi. Market Guide for Event Stream Processing. Technical report, Gartner Research, January 2022. URL: https://www.gartner.com/en/documents/4010467.

[48] I. Wang, E. Liri, and K. K. Ramakrishnan. Supporting IoT Applications with Serverless Edge Clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–4, November 2020. doi:10.1109/CloudNet51028.2020.9335805.

[49] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, and Frank Leymann. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Software: Practice and Experience*, 52(6):1454–1495, 2022. doi:10.1002/spe.3073.

[50] Ivana Podnar Žarko, Szymon Mueller, Marcin Płociennik, Tomasz Rajtar, Michael Jacoby, Matteo Pardi, Gianluca Insolvibile, Vasileios Glykantzis, Aleksandar Antonić, Mario Kušek, and Sergios Soursos. The symbIoTe Solution for Semantic and Syntactic Interoperability of Cloud-based IoT Platforms. In *2019 Global IoT Summit (GIoTS)*, pages 1–6, June 2019. doi:10.1109/GIOTS.2019.8766420.

6

Fog Native Architecture: Intent-Based Workflows to Take Cloud Native Towards the Edge

As the final chapter before the conclusion, this chapter takes a peek beyond the cloud to investigate how to help developers deploy applications that run on a mixture of cloud and edge resources. As such, it relates the final research question "How to adapt the cloud native paradigm to the cloud-edge continuum of the fog?" Both the research in previous chapters and the cloud native paradigm itself make a number of assumptions about the underlying infrastructure that do not hold true in an environment of mixed cloud and edge resources. This chapter investigates these issues in-depth and proposes an architecture to enable the cloud native experience in the edge. Although this chapter marks the end of the research presented in this dissertation, it is intended to be the beginning of a new line of research to facilitate service orchestration in the fog.

⋆ ⋆ ⋆

M. Sebrechts, B. Volckaert, F. De Turck, K. Yang and M. AL-Naday

Published in IEEE Communications Magazine, August 2022.

Abstract The cloud native approach is rapidly transforming how applications are developed and operated, turning monolithic applications into microservice applications, allowing teams to release faster, increase reliability, and expedite operations by taking full advantage of cloud resources and their elasticity. At the same time, "fog computing" is emerging, bringing the cloud towards the edge, near the end user, in order to increase privacy, improve resource efficiency, and reduce latency. Combining these two trends, however, proves difficult because of four fundamental disconnects between the cloud native paradigm and fog computing. This article identifies these disconnects and proposes a fog native architecture along with a set of design patterns to take full advantage of the fog. Central to this approach is turning microservice *applications* into microservice *workflows*, constructed dynamically by the system using an intent-based approach taking into account a number of factors such as user requirements, request location and available infrastructure and microservices. The architecture introduces a novel softwarized fog mesh facilitating both inter-microservice connectivity, external communication, and end-user aggregation. Our evaluation analyses the impact of distributing microservice-based applications over a fog ecosystem, illustrating the impact of CPU and network latency and application metrics on perceived Quality of Service of fog native workflows compared to the cloud. The results show the fog can offer superior application performance given the right conditions.

6.1 Introduction

The cloud native paradigm advocates for developing applications and network services to run intrinsically in the cloud, rather than merely transitioning to it [3, 8]. The objective is to realize applications at scale and provide capabilities including dynamic scaling, automatic recovery and seamless roll-out. This requires turning monolithic applications into *microservice applications* [5] by decomposing them into self-contained components interconnected by Application Programming Interfaces (APIs). The added complexity of managing microservices has been widely studied [4, 13, 1]. The considerable increase in message exchange between microservices, however, has been largely overlooked because it has not posed a major challenge given cloud providers' tight control over internal network bandwidth and latency.

At the same time, more and more companies are combining cloud applications with edge computing [6]. The Netflix Open Connect program, for example, invites ISPs to place Netflix caching servers in the edge, in order to increase user experience and decrease strain on the network. These capabilities are opened up to a much broader industry by fog computing, which extends the cloud towards the edge. This creates a new economic market where even small players can run applications over a mixture of cloud and edge resources. This enables application developers to increase privacy [15] and reducing latency [11], and helps ISPs to ensure more efficient resource usage. On

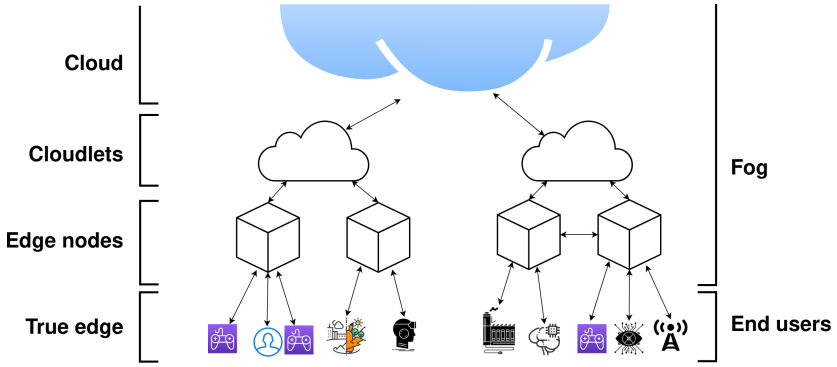

Figure 6.1: The OpenFog reference architecture showing a three-tier fog connected by a softwarized network controllable by management functions.

a technical level, this is enabled by providing both cloud and edge resources in a unified platform such as the OpenFog reference architecture [10] shown in Figure 6.1.

Taking full advantage of the fog requires an approach similar to cloud native paradigm. Naively applying this paradigm and tools to the fog, however, results in inefficient resource utilization and sub-optimal compute distribution; potentially negating the proximity benefits of the fog. These possibly counter-productive results are caused by a number of cloud native assumptions which do not always hold true in the fog.

- The cloud is relatively homogeneous and seamlessly hides the specifics of the underlying infrastructure from the end user, while the fog is inherently heterogeneous, accompanied with **operational complexity**.
- While clouds have an overabundance of relatively cheap resources, **resource constraints** become more apparent closer towards the edge of the fog.
- While clouds offer reliable low-latency and high-bandwidth communication between internal nodes, fog nodes are fully distributed and connected by a network with **highly variant latency and bandwidth**.
- The cloud is central with a relatively limited number of geographical locations, while the fog is **dispersed** with a much higher number of geographical compute locations offered by edge tiers.

Current efforts applying cloud native technologies to the edge do not fully address these issues. KubeFed, for example, allows creating federations of Kubernetes clusters, but is focused on the cloud instead of the fog. KubeEdge aims to shrink Kubernetes to fit on edge nodes, but does not address the fog's operational complexity and dispersion.

This paper identifies four fog native challenges and proposes an architecture that instigates a paradigm shift by defining applications as *microservice workflows*, constructed dynamically us-

ing *intent-based matching* of user requirements. This microservice workflow is an evolution from Service Function Chaining of Virtual Network Functions (VNFs), addressing its limitation of hard dependencies. This approach also provides better alignment with cloud native norms and offers uniform representation of 'logic-execution' elements in the fog. Moreover, the architecture introduces a novel *fog mesh* enabling seamless internal/external communication and flexible grouping of end-users. The architecture is evaluated in terms of distributing workflows over the fog to provide a baseline assessment on the impact of heterogeneity as well as the characteristics of microservices in a workflow on QoS and resource utilization.

This paper starts off by identifying four disconnects between cloud native and the fog in Section 6.2. Section 6.3 introduces the fog native architecture addressing these challenges. Section 6.4 outlines a pathway towards implementation through a sample use case. Section 6.5 provides a baseline assessment of workflow performance in the fog. Finally, Section 6.6 draws conclusions and outlines future work.

6.2 Disconnect between cloud native and the fog

This section identifies four key incompatibilities between cloud native and the fog. They stem from four assumptions about the underlying infrastructure and context that do not hold true in the fog.

6.2.1 Operational Complexity

Clouds present themselves as relatively homogeneous offerings, allowing developers to reason about what products and infrastructure to use for building an application. As a result, cloud applications are designed as rigidly connected microservices described using desired-state models [8]. Although these descriptions often use over-simplified assumptions of underlying operational complexity, this is not an issue in the relatively homogeneous-looking cloud.

The fog, however, is inherently heterogeneous from a developer standpoint. Providers cannot abstract away the underlying complexity because nodes have varying capabilities and availability of functionality is highly dependent on location, fog tier, and current resource usage [6]. Designing applications for a common denominator risks losing out on an untapped wealth of useful-but-not-ubiquitous features of fog nodes. Moreover, since resources in the fog are not infinitely scalable, design-time assumptions about infrastructure availability might not hold true anymore when the application is deployed.

6.2.2 Resource constraints

Clouds offer the illusion of infinite capacity at a relatively cheap price. Consequently, cloud native technologies do not necessarily optimize resource usage. Service meshes, for example, typically duplicate the number of containers needed to run a microservice application [4]. These meshes are

dedicated infrastructure layers that manage communication between microservices to improve observability, control and security of inter-microservice communication. They commonly use a "sidecar" approach in which each containerized microservice is accompanied by a containerized proxy which acts as an intermediate in all communications for that microservice. This approach works without code changes to the microservices themselves, but introduces significant resource overhead.

These inefficiencies are generally tolerable in a cloud environment with abundant resources. The fog, however, has strict constraints on resource consumption [6] that may not tolerate such waste.

6.2.3 Latency

Clouds offer reliable internal networks which enable low-latency and high-bandwidth communication between nodes. As a result, many cloud schedulers do not consider inter-service dependencies and network latency when placing services because their impact on performance is often negligible. This is not the case in the fog, however, due to its heterogeneous and turbulent internal network latency. As the evaluation in Section 6.5 shows, not taking into account inter-service dependencies and network parameters during scheduling of microservices in the fog results can negate much of the locality benefits of the fog.

A second issue arises in common cloud native patterns such as API gateways [13], which sit between a client and a microservice application, acting as the ingress endpoint for all external connections [1]. This pattern enables microservice applications to use asynchronous communication internally, and centralizes concerns such as compression, response aggregation and authorization [13]. However, given the aim of the fog is to bring compute closer to the edge, centralized gateways are antithetical to it, negating the latency [6] and privacy [15] benefits of keeping data and compute at the edge.

6.2.4 Dispersion

The cloud provides a relatively limited number of geographical compute locations, often called "regions". As a result, developers normally manually plan the geographical distribution of their microservices based on (predicted) user demand.

The fog, however, has a very high number of geographical compute locations [6], making it difficult to manually select where an application needs to run. Moreover, automatically distributing fog native applications based on individual end user requests can be challenging because of the sheer volume and diversity of them. Although this might be feasible for relatively static requirements such as a home automation system, it falls short at the scale and variability of applications such as video streaming and social networking.

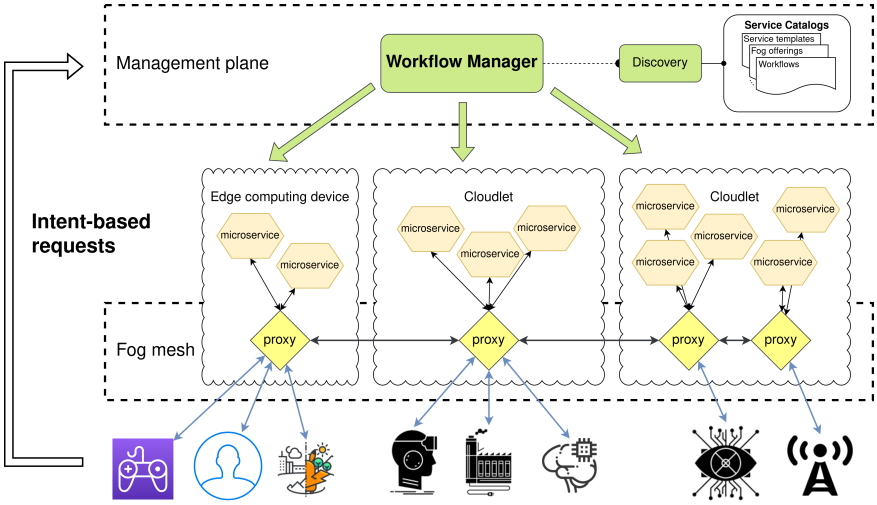

Figure 6.2: A fog native architecture showing a fog mesh supporting both internal and external communication, and a workflow manager using a discovery service to design and deploy microservice workflows based on user requests.

6.3 Designing a fog native architecture

This section introduces a number of fog native design patterns which fit together into an architecture that tackles the aforementioned challenges of bringing the cloud native paradigm to the fog.

6.3.1 Overview

The proposed fog native architecture instigates a paradigm shift where developers no longer define *what* should be deployed. Instead, they define *the desired behavior* of the application using *intents*. The system then dynamically composes and deploys *workflows of microservices*, taking into account a number of factors such as user location, network topology, available infrastructure, and existing services. This *intent-based workflow construction* permits much larger flexibility than the traditional desired-state approach because the orchestrator can change application topology and interchange application components depending on where user demand is for certain functionality and what infrastructure is available at that location.

Figure 6.2 shows an overview of the proposed fog native architecture, consisting of three conceptual components.

- The ***discovery service*** is a registry of individual microservice templates enriched with metadata about their functionality, characteristics and dependencies. This registry also tracks the offerings of the fog provider and the functionality of already deployed microser-

vices and workflows.

- The ***workflow manager*** responds to user requirements and either identifies a running workflow or uses intent-based workflow construction, explained further in Section 6.3.2, to create a new workflow from scratch to satisfy the user request.
- The ***fog mesh*** enables communication both between microservices, and to end-users. It consists of a set of interlinked service proxies, each of which provides softwarized network services to a regional cluster of microservices, saving up valuable resources by removing the one-to-one relationship between proxy and microservice.

The remainder of this section explains in detail the four key innovations of this fog native architecture compared to a traditional cloud native approach.

6.3.2 Intent-based workflow construction

As Section 6.2.1 explains, the fog's heterogeneity escalates the complexity for developers to describe their application's desired state. To tackle this challenge, we propose to dynamically create workflows based on user *intents*. This gives the system flexibility to dynamically update the desired state based on user demand, location and available infrastructure. Following this approach, user requests provide a description of the desired functionality, constraints, and tolerances. Developers advertise microservice templates annotated with rich metadata describing the functionality of each component, its infrastructure requirements and its dependencies. End users request certain functionality, for example using semantic-based addressing similar to that proposed by Al-Naday et al. [2]. The system parses this request and matches the intents with microservices, active workflows, fog offerings and VNFs. The system then either directs the request towards an existing workflow or dynamically constructs one that meets the demand. The resulting workflow can consist of microservices, VNFs, and XaaS offerings.

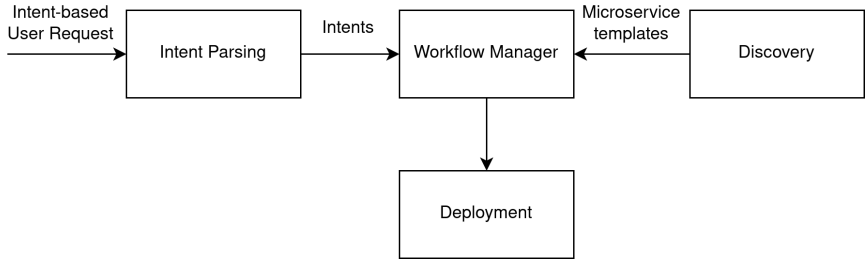

Figure 6.3: Intent-based construction of a new workflow.

As shown in Figure 6.3, when constructing a new workflow, the workflow manager selects a number of microservices and offerings using a matching logic which takes into account the locality of the end-user and workflow constraints in order to ensure the required QoS. The resulting workflow takes the form of a desired state model that contains multiple connected components such as

microservices and offerings. This model is delegated to lower-level orchestrators such as Kubernetes and/or deployed as VNFs. As a result, the practical implementation of requested functionality is different depending on the geographic location of the end-user, available infrastructure, and available microservices. Note that this work is focused on laying the foundations of intent-based workflow construction; leaving optimization solutions for future work.

6.3.3 Fog mesh providing inter-microservice connectivity

Internal connectivity in a cloud native environment is facilitated by a service mesh. As stated in Section 6.2.2, the current generation of service meshes are not well adapted to the resource constraints of the fog. Addressing this challenge requires removing the one-to-one relationship between sidecar proxies and microservices, and adding regional awareness from a network standpoint. The resulting fog native service mesh, "fog mesh", automatically groups microservices into a number of regional clusters based on network constraints. For example, microservices which are close to each other from a latency and network bandwidth perspective could share a single sidecar proxy. This vastly reduces the overhead required for the sidecar approach. Moreover, using this fog mesh, inter-microservice communication can use performant, not necessarily user-friendly, protocols. This mesh can be implemented as a flexible network of decentralized network functions.

6.3.4 Fog mesh providing external connectivity

The API gateway pattern is difficult to implement in the fog due to the heterogeneous network latency and possible spread of workflows over multiple regions, as explained in Section 6.2.3. Therefore, this architecture automatically distributes the API gateway functionality by merging it into the fog mesh. Each fog mesh proxy acts as an ingest point for the microservices connected by the proxy, providing service-based handling of internal-external communication, automatically configured based on local needs. This effectively merges the concepts of "service mesh" and "API gateway" into a single "fog mesh", which provides this functionality in a distributed manner.

This has two advantages: firstly, it regionally distributes API gateway functionality automatically based on microservices and network constraints as explained in Section 6.3.3. As a result, user requests and responses can be handled by the gateway in the region closest to the user, without the need for redirection to a central API gateway. Secondly, it removes the need for an additional service: fog mesh proxies handle both internal and external communication. This results in lower resource usage overhead.

Additionally, to fully utilize the potential for optimized communication, responses from microservices to end-users do not leave the fog mesh from the proxy closest to the last microservice, but from the proxy closest to the client. This fog mesh then translates the communication into a protocol tailored to the end user device, such as HTTPS in case of a browser. Notably, narrowing down the requests admitted by a proxy to those targeting the proxy's microservices combined with having a generally smaller number of users by virtue of locality is foreseen to incur a manageable state in the proxy.

6.3.5 Fog mesh providing end-user aggregation

In a cloud environment, the geographical distribution of an application is often manually designed by the application developer based on historic records of use and availability of cloud regions. This is infeasible in the fog, however, due to its high number of geographical compute locations as outlined in Section 6.2.4. Thus, a fog native scheduler is needed which distributes a microservice workflow based on end-user demand. Automating this distribution, however, is non-trivial due to the higher dispersion of users, causing higher demand variation and thus increased pressure on the fog scheduler.

To address this challenge, this architecture includes the novel design pattern of aggregating end-users and their workflow requests by the fog mesh. This allows the scheduler to make decisions on aggregations of end-user requests instead of individual requests, lowering the demand for scheduling decisions. Since, as explained in Section 6.3.4, fog mesh proxies act as the ingress point for local end-user requests, they have the required information to aggregate user requests for similar functionality into regional groups.

This, however, means that the end-user from a scheduling perspective is an aggregate and not the *actual* end-user. Since the fog mesh uses the scheduling information in order to manage communication, it will only be able to manage the connection up to the component acting as the aggregator. To ensure the aggregate component knows how to manage the connection to the end-user, this pattern introduces a *response-path token* uniquely identifying the end-user. This way, a workflow which, from a scheduling perspective, has a single end-user, can fan-out to an aggregate of nearby users.

6.4 Example: decision-support in the fog

This section outlines a pathway towards system-level implementation of the architecture, through an example use case of UAV-based disaster management; enabling a dynamic decision-support system based on live drone feeds for aiding first responders. Autonomous drones observe the incident area, interpret the data and provide an action list prioritized on urgency or danger to responders as shown by Moeyersons et al. [9].

Due to the unexpected nature of most incidents, proactively designing and deploying decision support pipelines on location is not possible. Due to the high bandwidth and low latency requirements of the decision support pipeline, running them on a centralized data center will negatively affect both the end user experience and strain the network resources. Therefore, this requires a system that dynamically designs and schedules local workflows based on responders' needs (i.e. *intents*)

Figure 6.4 illustrates the example using the proposed fog native architecture. A subset of fog mesh proxies may already be active at the responders' site while others have yet to be instantiated at the incident site. The proxy's implementation can build on experience gained from sidecar and

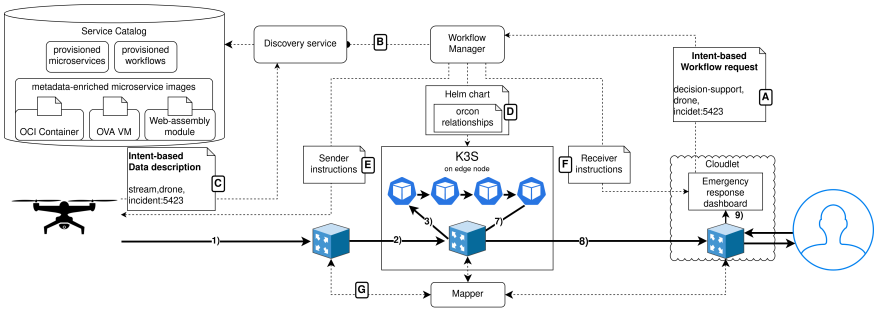

Figure 6.4: Overview of the proposed fog native architecture enabling drone-based decision support for crisis response teams through matching of responders intents with offered microservices and data.

API gateway technologies. The workflow manager and discovery service are placed in the network, possibly at the responders' site, having latency-bound communication with both end-user proxies. Notably, this example assumes a single instance of each management service controls the fog resources of both sites of interest; otherwise, distributed instances of each service might be needed.

In this example behavior, an emerging incident may trigger an intent-based workflow request by the responders' proxy (A), describing required tasks and data, including video feeds. The workflow manager interacts with the discovery service (B), to identify existing components and data. A subset of microservices may already be available, while others - such as the drone feed - are yet to be established. Upon arrival to the scene, the drone uses a fog proxy to advertises an intent-based description of its video stream to the discovery service (C). The latter informs workflow manager to complete the construction of the workflow. The deployment of the workflow is delegated to lower-level orchestrators. In this example, the workflow manager contacts a K3S edge node and uses a Kubernetes relationship orchestrator [12] to deploy an undistributed composition of microservices (D). After a successful deployment, the workflow manager instructs the drone to start transmitting the video stream (E) to be processed by the workflow, i.e. by the microservices following their dependency map in a hop-by-hop fashion. The last microservice delivers the decision support outcome to the responders' dashboard after it completes.

6.5 Evaluation

This section analyzes the impact of distributing workflows over a fog ecosystem using the discrete event simulation framework simmer [14]. The analysis illustrates the impact of internal network latency and application metrics on the perceived Quality of Service (QoS) of workflows distributed over the fog compared to *undistributed workflow allocation* in cloud systems. We present our results in terms of the *latency residual budget*, corresponding to QoS by measuring the difference between the latency threshold of a workflow, specified by the end-user, and the observed response

time. A negative value indicates a violation of the latency agreement.

A fog native network is modeled as a set of *fog nodes* overlaid on top of a softwarized routing network. Traffic per workflow is modeled as a set of simmer *trajectories* starting from the user to the fog node of the first microservice; and ending from the fog node of the last microservice back to the user. The CPU capacity of each node depends on the node's tier, with cloud nodes having the highest CPU capacity and edge nodes having the lowest. Similarly, network links are characterized by their bandwidth capacity (Mbps) and length (Km) with core links having the largest bandwidth and longest distance and edge links having the smallest and shortest counterparts. Propagation latency and the queuing counterpart at each routing node are calculated using link attributes, current state and data size. The processing latency of the deployed microservices are derived from the CPU capacity, current workload of a fog node, and task size. The total response time is then calculated as the additive accumulation of all latencies, between 'user-to-first-microservice' and 'last-microservice-to-user'.

For application workflows, the evaluation considers two forms of dependency maps: *Chain* and *Hub and Spoke* (H&S). In a *Chain* map, microservices are serially related to each other; whereas in a H&S map, the first and last microservices are *hubs* and intermediary ones are *spokes*. A workflow may either be distributed (i.e. microservices assigned) over multiple fog nodes, hence classified as *Distributed* or all corresponding microservices are assigned to one fog node and so deemed *Undistributed*. Moreover, each workflow is characterized by a *latency budget* indicating the maximum tolerable response time. Each microservice has a task size measured in number of CPU cycles, and input and output data measured in megabytes.

The simulation assumes 100 workflows offered in the network, each of which consists of 5 microservices selected from a catalog of 1000. Each workflow has either *Chain* or *H&S* dependency map. The CPU and data specification per microservice is defined per scenario. For the network, we consider the topology of the AT&T MPLS network of 25 nodes and 114 links [7]. It assumes a $3-$Tier fog network with: tier-0 central cloud (2 nodes), tier-1 the smaller *cloudLets* (4 nodes) and tier-2 the highly constrained edge (8 nodes). The fog nodes in each tier are placed randomly in the network. In all the results, the CPU and bandwidth capacities of any tier are approximately 10% equivalent of the upper tier. Finally, the simulation assumes each switching node to connect between 1000 and 4000 end-devices, generating requests for workflows at a rate of approximately 1500 requests per second.

6.5.1 Latency vs. distributability

This evaluation analyzes the interplay between workflow dependency and infrastructure distribution, and the impact on latency perceived by end-user.

Figure 6.5 shows the latency residual budget when varying the distribution of the fog infrastructure, extending from the typical central cloud to a hierarchically distributed fog. The results are shown for both undistributed workflows and workflows distributed randomly over multiple fog

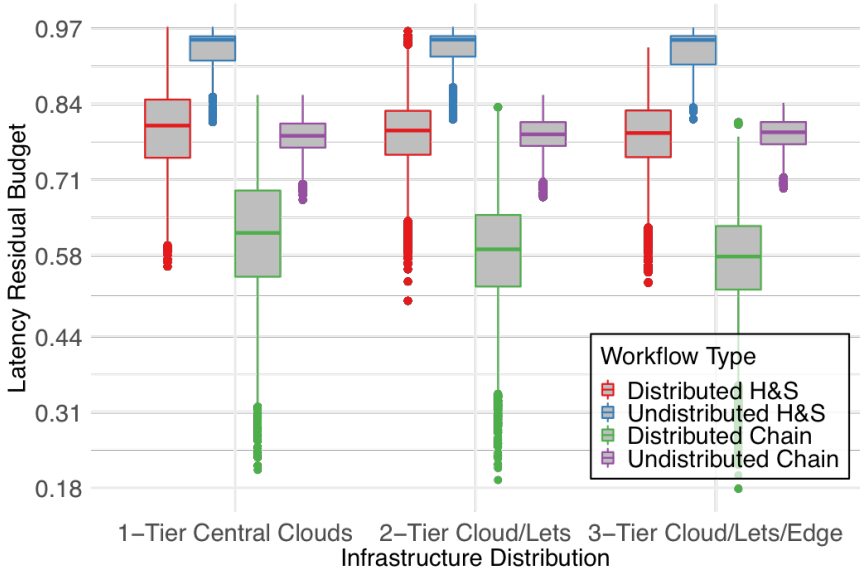

Figure 6.5: The latency residual budget when varying the fog infrastructure from central cloud to hierarchical fog.

nodes. The observed latency residual budget for distributed workflows is lower than for undistributed variants, illustrating the impact of network latency on overall response time. Notably, the average residual budget of H&S workflows is approximately 20% higher than that of chain workflows, showing improvement as a result of parallelizing microservice execution. Moreover, the residual budget for distributed workflows decreases as the infrastructure changes from central to distributed. This increased response time is caused by

- additional communication latency from distribution of the workflow over a larger number of fog nodes, and
- increased computation latency from the lower CPU capacity of edge infrastructure.

Interestingly, undistributed workflows in a hierarchical fog perform no different from their counterparts in central clouds, showing the reduction in *communication* latency is countered by increased *computation* latency.

6.5.2 Latency vs. application metrics

Figure 6.6 shows the latency residual budget when having variant task and data size. The results show the response time for workflows of large sized data, (approximately 2-4 megabyte) is on average 10-25% higher than for small data (approximately 0.5-2 megabyte), irrespective of the task size. Nonetheless, workflows with large task size have, on average, a higher response time

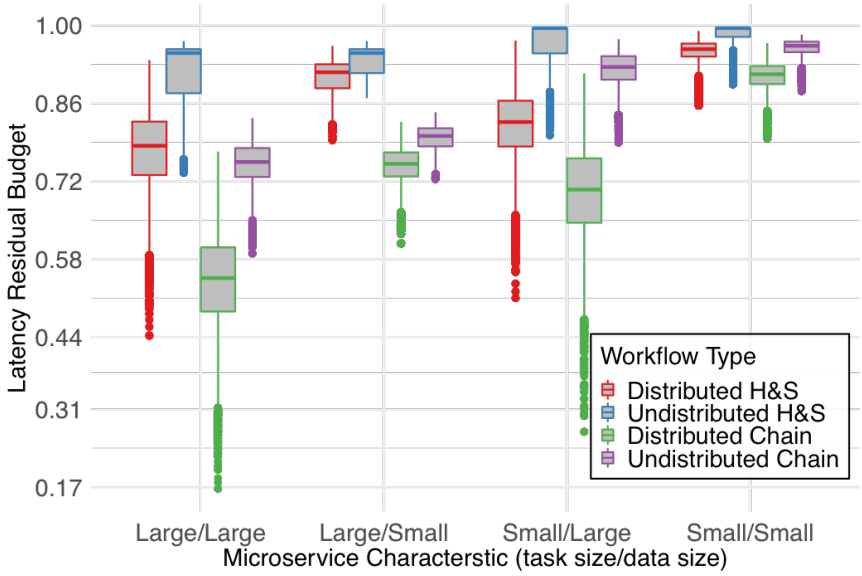

Figure 6.6: The latency residual budget for variant task and data size in a 3-tier fog.

by approximately 5-20% compared with workflows of small task size, for the same data size. This reveals the significance of communication latency when having to transmit large volumes of data. The dependency map and distribution also impact the perceived response time. H&S workflows incur the quickest response time, even when comparing distributed ones to undistributed chains. This shows the effect of parallel microservice execution and the interplay with CPU and link bandwidth capacities. Although the last microservice in a H&S workflow waits for all the intermediaries to complete, the combined execution and waiting time remains smaller than that in undistributed chains. Although distributing a workflow reduces residual latency, the budget is not exceeded, which means it can be a valid option to reduce workload congestion by spreading compute load.

6.6 Conclusion

Operational complexity, stringent resource constraints, varying internal network latency, and high granularity of geographic regions make the fog inherently incompatible with the cloud native paradigm. To address this challenge, this work proposes a fog native architecture along a set of design patterns to facilitate flexible and dynamic provisioning of microservice-based applications over the heterogeneous fog. Using intent-based workflow construction, applications are composed of loosely-dependent microservices selected to best match user requirements. A novel fog mesh enables microservice grouping under one proxy, seamless user-microservice and inter-microservice communications, and request aggregation. To illustrate a pathway towards implementation of the architecture, this article describes an example use case of drone-based decision

support for first responders in the fog. Evaluation shows the impact of running microservice-based applications in a fog ecosystem, confirming, for example, network latency plays a bigger part in distributed workflow response time in the fog compared to the cloud.

Future work is foreseen to provide a prototype of the fog mesh and further investigate algorithms, optimizations and implementations for translating intents into desired state models. It will also further investigate management of data at rest in the fog.

Bibliography

[1] Akhan Akbulut and Harry G. Perros. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing*, 23(6):19–27, November 2019. doi:10.1109/MIC.2019.2951094.

[2] Mays Al-Naday and Irene Macaluso. Flexible Semantic-based Data Networking for IoT Domains. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, June 2021. ISSN: 2325-5609. doi:10.1109/HPSR52026.2021.9481800.

[3] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In Antonio Celesti and Philipp Leitner, editors, *Advances in Service-Oriented and Cloud Computing*, Communications in Computer and Information Science, pages 201–215, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-33313-7_15.

[4] Amine El Malki and Uwe Zdun. Guiding Architectural Decision Making on Service Mesh Based Microservice Architectures. In Tomas Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 3–19, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-29983-5_1.

[5] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-Native Applications. *IEEE Cloud Computing*, 4(5):16–21, September 2017. doi:10.1109/MCC.2017.4250939.

[6] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 98:27–42, November 2017. doi:10.1016/j.jnca.2017.09.002.

[7] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, October 2011. doi:10.1109/JSAC.2011.111002.

[8] Tom Laszewski, Kamal Arora, Erik Farr, and Piyum Zonooz. *Cloud Native Architectures: Design High-Availability and Cost-Effective Applications for the Cloud*. Packt Publishing, 2018.

[9] Jerico Moeyersons, Pieter-Jan Maenhaut, Filip De Turck, and Bruno Volckaert. Aiding First Incident Responders Using a Decision Support System Based on Live Drone Feeds. In Jian Chen, Yuji Yamada, Mina Ryoke, and Xijin Tang, editors, *Knowledge and Systems Sciences*, Communications in Computer and Information Science, pages 87–100, Singapore, 2018. Springer. doi:10.1007/978-981-13-3149-7_7.

[10] OpenFog Architecture Workgroup. OpenFog Reference Architecture for Fog Computing, February 2017.

[11] István Pelle, János Czentye, János Dóka, and Balázs Sonkoly. Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS. In *2019 IEEE 12th International Conference*

on Cloud Computing (CLOUD), pages 272–280, July 2019. doi:10.1109/CLOUD.2019.00054.

[12] Merlijn Sebrechts, Sander Borny, Tim Wauters, Bruno Volckaert, and Filip De Turck. Service Relationship Orchestration: Lessons Learned From Running Large Scale Smart City Platforms on Kubernetes. *IEEE Access*, 9:133387–133401, 2021. doi:10.1109/ACCESS.2021.3115438.

[13] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Microservices Anti-patterns: A Taxonomy. In Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Patricia Lago, Manuel Mazzara, Victor Rivera, and Andrey Sadovykh, editors, *Microservices: Science and Engineering*, pages 111–128. Springer International Publishing, Cham, 2020. URL: https://doi.org/10.1007/978-3-030-31646-4_5.

[14] Inaki Ucar, Jose Alberto Hernandez, Pablo Serrano, and Arturo Azcorra. Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping. *IEEE Communications Magazine*, 56(11):145–151, 2018. doi:10.1109/MCOM.2018.1700960.

[15] Chunyi Zhou, Anmin Fu, Shui Yu, Wei Yang, Huaqun Wang, and Yuqing Zhang. Privacy-Preserving Federated Learning in Fog Computing. *IEEE Internet of Things Journal*, 7(11):10782–10793, November 2020. doi:10.1109/JIOT.2020.2987958.

7

Conclusions and perspectives

This concluding chapter reflects on the proposed solutions in the broader context of this dissertation and identifies interesting open challenges.

7.1 Reflecting on research questions

Each of the solutions proposed in the previous chapters relate to one of the research questions described in Chapter 1. This section shows how these solutions address the research questions in the broader context of this dissertation. At the time of writing this dissertation, some of the research presented has been published over four years ago. As such, the benefit of hindsight gives the unique opportunity to reflect on some of the lessons learned since initial publication. This section also peeks into how the presented research can play a role in future developments in the field of service orchestration in the cloud and fog.

7.1.1 Deciding what should be deployed

The orchestrator conversation presented in Chapter 2 presents an answer to the question of "How to encapsulate and reuse system administrator's knowledge about when to deploy what?". The chapter breaks this challenge down into the more fundamental question of "how to let system administrators create new abstractions". Since it is difficult to create new abstractions using desired state models, the chapter proposes a collection of independent agents which translate higher-level models into lower-level ones.

Kubernetes addresses this challenge in a similar way using "controllers". These are services developed using regular programming languages that translate higher-level abstractions into lower-level ones. The difference between this approach and the orchestrator conversation lies in the centralized nature of controllers. Kubernetes itself is a centralized orchestrator with a single source of truth: the Kubernetes API. As such, controllers follow this model by extending this central API with new abstractions. A single controller can manage as many instances of this abstraction as needed. A single controller instance might, for example, manage three separate Spark clusters. In the orchestrator conversation, on the other hand, there is no single source of truth: each service agent has a limited view on reality. As such, abstractions are added in a decentralized manner. The abstraction provided by an orchestration agent can thus only be used by entities directly connected to it. To manage three separate Spark clusters, three orchestration agents need to be deployed. Herein lies a major downside of agent-based orchestrators such as Kubernetes: the overhead of the control plane. Chapter 4 shows that managing relationships using a centralized controller incurs much less overhead than using an agent-based approach. Moreover, in a microservice application, the memory usage of an agent might come close to the overhead of the microservice it manages [3].

Looking into the future, however, it is probably unwise to discount an agent-based approach entirely. As Chapter 6 discusses, centralized systems common in the cloud become challenging in the cloud-edge continuum of the fog. As such, an agent-based approach might be the only viable solution in such an environment. Nevertheless, the overhead issues of an agent-based approach would still need to be addressed, since resources become more limited as workloads move towards the edge.

7.1.2 Deciding when to run management actions

The reactive pattern presented in Chapter 3 presents an answer to the question of "How to encapsulate and reuse system administrator's knowledge about when to perform which management actions?" Since it is challenging to reuse parts of traditional lifecycles, the reactive pattern proposes an event-based system where lifecycles emerge from chaining conditional handlers. The chapter highlights the charms.reactive SDK which implements this pattern on top of the Juju orchestrator.

The evaluation in Chapter 3 shows that after two years since its initial release, charms.reactive did indeed create an ecosystem of code sharing and collaboration between developers creating charms. At the time of writing this thesis, seven years after inception of this framework, Canonical is still maintaining and using it. In 2019, however, it has been officially superseded by the "Charmed Operator Framework". Interestingly, this new framework continues the idea of event-based custom lifecycles in order to facilitate knowledge sharing between Charm developers [1]. A common complaint of the reactive approach is that its complexity makes it difficult to wrap your head around. It is not always clear, for example, what specific workflow will emerge from certain preconditions. It was not uncommon for developers to be surprised by the emerging be-

havior of their code, which is not always a good thing. One possible way to address this would be to dynamically generate the emerging workflow during design time, to give developers a better understanding of it. This is difficult to do with the charms.reactive framework because the pre- and post-conditions of a handler are only truly known at runtime. Requiring developers to declare these at design time would make this easier, although it is an additional limitation on the flexibility of a handler.

Looking into the future, it seems the event-based approach of the reactive pattern still has merit to promote code reuse. Although Kubernetes has a growing ecosystem of controllers and tools to create them, code sharing between controllers is still difficult. The operator SDK, for example, is a common toolset to create controllers. Although it adds a fair amount of functionality to facilitate creating controllers, this functionality is focused on working with Kubernetes itself. Very few examples exist of application or microservice-specific functionality being embedded in libraries for reuse.

7.1.3 Deciding how to connect microservices

The "orcon" orchestrator presented in Chapter 4 aims to address RQ 3: "How to encapsulate and reuse system administrator's knowledge about composing microservice applications and (re-)configuring their internal dependencies, in a way that fully integrates into a cloud native ecosystem?" It extends the Kubernetes API to allow users to model microservice dependencies and it automatically manages these dependencies.

The evaluation in that chapter shows this approach has much less overhead than cloud modeling languages such as Juju. The overhead of orcon is still larger than that of Helm, however, indicating more optimization could be achieved. The evaluation also shows the original Kubernetes API is not obscured by orcon, so it integrates into the existing Kubernetes ecosystem. One example of this integration is that it is possible to use Helm to define and modify orcon relationships. This makes it possible for users to encapsulate and share applications using orcon. It is also possible to define and modify orcon relationships using Argo CD, in order to set up a robust CI/CD system for managing an application using orcon.

7.1.4 Deciding how to process cross-domain streams

The "Plumber" platform presented in Chapter 5 aims to address the fourth research question: "How can multiple independent parties collaboratively create serverless streaming pipelines?" It extends the Kubernetes API to allow multiple independent parties to collaboratively model serverless streaming pipelines and it automatically creates and manages these pipelines.

The evaluation of Chapter 5 shows multiple independent parties can indeed collaboratively create these pipelines. Moreover, it shows there is minimal data-plane overhead, and updating a topology is achieved in 12 seconds without any data loss. Just like orcon, Plumber is a Kubernetes-native framework and thus integrates into the wider ecosystem with tools like Helm and Argo CD.

7.1.5 Bringing cloud native to the fog

The fog native architecture presented in Chapter 6 relates to RQ 5: "How to adapt the cloud native paradigm to the cloud-edge continuum of the fog?" It addresses four key incompatibilities between the cloud native paradigm and the fog: increased operational complexity, more stringent resource constraints, highly variant network latency and bandwidth, and high geographic dispersion. To solve these, the architecture proposed a paradigm shift where developers no longer define *what* should be deployed. Instead, they define *the desired behavior* of the application using intents.

This chapter includes an evaluation that analyzes the impact of distributing workflows of microservices over a fog ecosystem. This confirms, for example, that network latency plays a much bigger part in distributed workflow response time in the fog compared to in the cloud. The intent-based approach is a radical change from the work presented in previous chapters. Whereas the previous chapters solved service orchestration problems using declarative desired-state models, this chapter details how this approach is not feasible anymore in the fog. As such, although this chapter marks the end of the research presented in this dissertation, it is intended to be the beginning of a new line of research to facilitate service orchestration in the fog.

Looking at the future, however, it is clear that declarative desired state models will also play an important role in the fog. Not as the main user-facing API, but as an underlying intermediary format in machine-to-machine communication. As the architecture proposes, workflow managers translate intents into instructions to send to lower level orchestrators, which is what declarative desired-state models are made for. This is where the orchestrator conversation from Chapter 2 might come in. Its design is uniquely suited to the decentralized nature of the fog and it is focused around machine-to-machine communication. The same approach might be key to creating a truly decentralized fog orchestrator.

7.2 Future perspectives

7.2.1 Agent-based orchestrators in the fog

While centralized orchestrators work great in the cloud, they are less ideal for the distributed nature of the fog. As such, future work should look at how agent-based distributed service orchestration methods such as the orchestrator conversation proposed in Chapter 2 can be adapted to the fog.

- One key challenge in adapting agent-based orchestration to the fog is the overhead of agent-based approaches. A promising way to solve this might be to create on-demand agents using WebAssembly, similar to the on-demand Kubernetes controllers shown in [2].
- Addressing the cross-domain nature of the fog is equally important. One interesting approach to tackling this issue is to combine agent-based approaches with swarm algorithms

and promise theory. The agent-based approach can be used to create a federation of orchestrators. Swarm algorithms could be used to solve the inherent issues of having multiple independent agents operate without a single global view of the entire truth. Finally, promise theory could be used in order to ensure QoS and other metrics across such a network of independent agents.

- Another interesting challenge is in how to create a true fog ecosystem of infrastructure providers, service developers and orchestration providers.
- Given the large ecosystem which is emerging around the Kubernetes API, it is interesting to investigate whether it is possible to create a distributed agent-based system which provides the same API as Kubernetes.

7.2.2 Improving collaboration in the creation of Kubernetes controllers

Although Kubernetes has a growing ecosystem of controllers and tools to create them, code sharing between controllers is still difficult. The operator SDK, for example, is a common toolset to create controllers. Most of its functionality, however, is focused on making it easier to communicate with the Kubernetes API and to integrate into the Kubernetes control loop. It does not contain application-specific logic. The only knowledge captured in this library is about how to write controllers, not how to manage applications or services. This is common with other toolsets: very few examples exist of application or microservice-specific knowledge being embedded in libraries for reuse.

The reactive pattern proposed in Chapter 4 might be a good candidate to solve this issue. Although the successor of the charms.reactive framework can be used to manage applications running on Kubernetes, it is built on top of the Juju orchestrator which hides the Kubernetes API behind a new abstraction. Therefor, it might be interesting to investigate how to adapt this pattern to Kubernetes controllers without adding an additional abstraction on top of Kubernetes.

7.2.3 Secure service orchestration in the Fog

Moving services to the fog also considerably changes the security considerations of their service orchestrations. For example, ensuring physical security of devices in the edge is much more difficult than in the cloud. This has considerations for both infrastructure providers as infrastructure users. Providers need to ensure they can trust devices which join their platform and ensure that a physical compromise of a device does not mean their entire platform is compromised. Users, on the other hand, need to take into account that a physical compromise is much more likely and that the hardware can have varying levels of trust.

- Device attestation refers to a collection of approaches to ensure the hardware and software of a remote device is in a certain state. It can make sure that the software of a remote device has not been tampered with using approaches such as Trusted Platform Modules and Intel Software Guard Extensions. There is a need for novel solutions which integrate these

device attestation approaches with orchestrators such as Kubernetes. Device attestation approaches often employ chained trust where each layer of a system checks the integrity of the layer above it. The hardware root of trust is the lowest layer from which this process starts. As such, a second challenge concerning device attestation lies in the relationship between the root of trust and a user of the infrastructure. Depending on how and by whom the root of trust is configured, the trust level of a device might change drastically from the viewpoint of a user. As a result, service orchestrators should have mechanisms to present this information to the user and to take this information into account during scheduling decisions.

- The increased risk of physical compromise and varying levels of trust creates a second challenge: is it possible to run workloads on infrastructure that is not fully trusted? One possible way to address this issue is to look into confidential computing. Intel Software Guard Extensions, for example, make it possible to hide certain aspects of an application from even the lowest levels of the operating system and hypervisor. This can, for example, ensure encryption keys of a process are not visible to other entities. It can also run certain processes in a trusted enclave, ensuring that the entire process is not visible to other entities. The downside of these approaches is that they are resource intensive and require special hardware support. Novel approaches are required to integrate these technologies into service orchestrators. For example, an orchestrator could dynamically choose to use certain trusted computing methods for executing workloads based on the trust level and available features of the infrastructure. Since certain trusted execution solutions require code changes, this is another factor that should be taken into account during scheduling decisions.

7.2.4 Intent-based orchestration

Chapter 6 shows how intent-based orchestration is important to facilitate service orchestration in the fog. Although there is a lot of active research on using intent-based approaches in software defined networks, there are still a number of open challenges concerning orchestrating microservice applications using intents.

- The first challenge is in the structure and form of intent models for microservice applications. Novel research is needed to investigate what form these models should take. One approach might be to combine intent-based statements with existing desired-state approaches in order to find a middle ground between flexibility for the orchestrator and concreteness for the developer.

- The second challenge concerns how to translate intent models into concrete desired state models. Architecturally, an agent-based solution such as the orchestrator conversation could be extended with a new type of agent which does this translation. The question remains, however, how this agent gathers and processes the current state of available infrastructure and available demand in order to create a desired state model which im-

plements the desired intents. Artificial Intelligence for IT Operations (AIOps) might play a role in this in order to ensure the desired state models are optimized and take into account possible future changes to infrastructure and demand.

- Another important question is in how such an intent-based system would handle the inherent uncertainty of distributed infrastructure running highly heterogeneous workloads. Scheduling algorithms in a homogeneous cloud environment have the advantage to work with a complete picture of the current state of the infrastructure. This makes it relatively easy to predict the effect of a workload scheduling decision when the behavior of the workload itself is known. This is much less the case when a heterogeneous fog environment is combined with incomplete information about the current state of the system. This appears to be another challenge in which AIOps might play a significant role, in order to reason on an incomplete data set and predict the resulting behavior of decisions.

Bibliography

[1] Jon Seager. Juju | Libraries, April 2021. URL: https://juju.is/docs/sdk/libraries.

[2] Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, and Filip De Turck. Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI. 2022.

[3] Merlijn Sebrechts, Gregory Van Seghbroeck, and Filip De Turck. Optimizing the Integration of Agent-Based Cloud Orchestrators and Higher-Level Workloads. In Daphne Tuncer, Robert Koch, Rémi Badonnel, and Burkhard Stiller, editors, *Security of Networks and Services in an All-Connected World*, Lecture Notes in Computer Science, pages 165–170. Springer International Publishing, 2017. doi:10.1007/978-3-319-60774-0_16.