KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Deployment Efficiency and Data Security for the Cloud

Emad Heydari Beni

Supervisors:
Prof. dr. ir. Wouter Joosen
Dr. Bert Lagaisse

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

August 2021

Deployment Efficiency and Data Security for the Cloud

Emad HEYDARI BENI

Examination committee:
Prof. dr. ir. Robert Puers, chair
Prof. dr. ir. Wouter Joosen, supervisor
Dr. Bert Lagaisse, supervisor
Prof. dr. ir. Yolande Berbers
Prof. dr. ir. Yves Moreau
Prof. dr. Nigel Smart
Dr. Dimitri Van Landuyt
Prof. dr. ir. Filip De Turck
(Ghent University, Belgium)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

August 2021

Preface

Pursuing a PhD for me has been a roller coaster ride with failures & successes, as well as deaths & births. It is hard to believe that I am finally writing this section, and closing this chapter of my life. It is indeed the highest time to express my gratitude and remember those who helped me in this journey.

First and foremost, I would like to thank my supervisor, Wouter Joosen, for giving me the opportunity to have this unique experience. Your advice along the way has influenced my career fundamentally, and as a consequence, I could work with some of the smartest people I have ever met. You have made our research group, DistriNet, a place for innovation without unnecessary tensions typically found in other research groups. Academia needs more people like you.

Secondly, I want to thank Bert Lagaisse. You, as my coach and co-supervisor, have guided me throughout every step of my research. You were sometimes more than a coach to me. You gave me the opportunity and most importantly support to follow what I had a passion for. I hope you also consider this dissertation an achievement for yourself if you still can remember the very first draft of the WF-Interop paper. This accomplishment would not have been possible without you. Lastly, thanks to you I managed to have the most burger-intensive moments of my life and reach 90kg.

Furthermore, I am grateful to the other members of my jury for the interesting discussion and the valuable feedback: Yolande Berbers, Yves Moreau, Nigel Smart, Filip De Turck, and Dimitri Van Landuyt. I would also like to thank Robert Puers for chairing the jury and the PhD defense.

Next, I want to thank Eddy Truyen. Together we coached several Master's students, including Matthijs, Jordy, Stef, and Arno. Thanks to you I learned how we can conduct decent software and systems research in the scope of performance engineering. This legacy stays with me for the rest of my career. Lastly, our fruitful collaborations, especially with Jordy, have had an important impact on this thesis. May the Gods of Kubernetes bless us all.

I have started in office 05.33, stood on the shoulders of giants, and now finished in the same office. First of all, I want to thank Rinde who has inspired me in my early years and helped me feel at home in the office. Our legendary office mates were Rinde, Kristof (K), Jonathan, and Tung. With Alex and Hilaria, you all came to Poland for my wedding and made it a wonderful ceremony for me. No Pole has ever drunk that much vodka in one night; you guys have set a record. I would like to thank the current office mates as well. I want to thank Kristof (▲) for his investment tips as a crypto whale, Martijn for our collaborations and proofreading this text, Federico for being a symbol of someone who can eat a small cookie for lunch and yet survive, and Pieter-Jan for our ongoing research work.

Needless to say, many thanks go out to all (ex-)DistriNet colleagues, especially, Bart, Danny, Dominique, Mathijs, Giuseppe, Koen (J), Tobias, Vincent, Ansar, Victor, Vera, Laurens, Kim, Koen (Y), Willem, Davy, Jan Tobias, Sam, Job, Tim, Stefan, Stef, Ilias, Mario, Wilfried and many others that I probably miss their names. Moreover, I want to thank my Iranian friends, including Amin, Jafar, Sepideh, Ehsan, Majid (M), Majid (S), Nima, Omid, and the king of NLP, Erfan. In particular, I would like to thank Amin since he was always there to listen to my concerns and guide me whenever I had to make tough decisions. Lastly, I would like to thank my old friends at UAntwerpen who have been always my coolest friends: Ali, Leonard and Frank.

In addition, I am grateful to the administrative and technical staff of the department for their valuable support. Special thanks go out to Annick, An, Annelies, Ghita, Hilde, and the System Group team for their support. Super special thanks go out to Katrien who has been always there for me. When my father passed away, you were the one who could understand the depth of my sadness and talked to me for hours. You are a shining star in DistriNet.

I want to thank all my research collaborators and industry partners. Special thanks go out to Marco, Michael, Svetla, Abdel, Ren, Filipe, Anja, Paul, Ronny, Roberto, Toon, and many others.

To be positive today, I would like to thank Reviewer-2 who always complained and he never understood what I really meant. Next, I would also like to thank the company that declined my already-arranged research visit because of my nationality. No one is perfect.

When I was younger, in my early student days in Belgium, someone kindly helped me to continue my education. In fact, without her, I would probably not be here writing this thesis. Her name is Halleh Ghorashi, a well-known anthropologist. Halleh! Although I have never met you (nor talked to you), you have a special place in my heart, and you are my Daddy-Long-Legs.

Next, I would like to express my gratitude to my family and family in law. You have always supported me and have never given up on me when I was tired and grumpy. I wish my father, may he rest in peace, could have been here with us to celebrate this milestone in my life and career. He was one of the key motivating reasons for me to join academia and pursue a PhD. Moreover, I would like to thank my sister, Shiva, and her husband, Ramtin, for listening to my rants and numerous phone calls while having dinner. You are the best.

I would like to thank my mother for taking care of everything since day one in the 80s. Although I would not be able to achieve anything in my entire life without you and all I have is because of your endless support, education has put a geographical distance between us for almost 15 years; I am sorry. We live in an unfair world with many borders and discriminations. This thesis is the fruit of your 33-year-long effort. Thank you for everything!

Last but not least, I would like to thank my wife, Kasia. Thank you for your patience and for helping me to cope with my difficult days during the last few years. As a joke, I planned to write: *To my wife Kasia and my daughter Ava without whom this thesis would have been completed 2 years earlier*; however, I confess probably it would have been never finished without you. You have always supported me to think logically and make important decisions in my career. I hope someday I can do the same for you. I would like to dedicate this thesis to you, Ava, and our future baby girl.

– Emad Heydari Beni, October 2021

Abstract

Software service providers increasingly adopt the cloud computing paradigm because it provides on-demand access to a nearly unlimited pool of resources. This typically entails outsourcing their computation and data to the cloud through the deployment of their software services, storing business-sensitive data, and streamlining their processes on the cloud. Two of the core challenges that the industry faces concern with (i) how to achieve outsourced computation in an efficient and manageable manner through agile and optimised deployments, and (ii) how to protect sensitive data in untrusted environments, in particular the public cloud, and yet pragmatically preserve business functionalities. To optimise the deployments, cloud providers have been improving the computing infrastructure, e.g. by introducing container orchestration frameworks to improve the automation and agility of software deployments. Moreover, at the higher layers, software service providers have been migrating complex and compute-intensive software systems, such as engineering workflows, to the cloud to gain efficiency. To preserve the utility of protected data in the cloud, researchers have proposed many cryptographic techniques for search and computing on encrypted data.

However, software practitioners are confronted with several challenges. First, the elastic scaling of computing resources such as containers causes non-negligible delays, known as *cold start*, which prohibits agile and swift large-scale deployment of software services. This concern often appears in serverless computing, and in general auto-scaling systems, for applications with deadline-based service-level objectives (SLOs). There are many techniques for low-latency instantiation of resources at various layers of the infrastructure. However, the existing approaches for Kubernetes, the de facto standard in container orchestration, are still unable to fully mitigate the cold start problem. Second, distributed and compute-intensive applications, in particular engineering workflows, require tailor-made and repeatable deployments to decrease execution time in the cloud. The execution of this class of applications might take days, but the right composition of cloud resources can potentially reduce the

completion time to minutes. However, it is not trivial for engineering domain experts to decide upfront the amount and type of the required cloud resources and make cost-effective decisions for future iterations of their deployments. Third, the cryptographic techniques for search and computing on encrypted data on the cloud are diverse in terms of security, performance, and query expressiveness. Moreover, the underlying concepts and implementation details of these schemes are often complicated for application developers. Next to the error-prone process of choosing a scheme, integration of these data protection techniques into heterogeneous and polyglot software, e.g. microservice architectures, is a non-trivial task for security experts. Lastly, the ability to plug in cryptographic schemes in this scope is an important aspect since new cryptographic constructions with new properties are being presented frequently. These issues require a certain degree of cryptographic agility.

The goal of this dissertation is to address the above-mentioned challenges in order to facilitate the process of outsourcing computation to the cloud via efficient deployments, and pragmatically store data on the cloud using advanced protections. To achieve this, our contributions are threefold: (i) we present and evaluate three approaches to reduce the cold start latency during elastic scaling of containers in Kubernetes. To achieve this, we present various techniques such as library sharing, pre-creating networking resources, and employing imperative configuration management unlike the existing declarative configuration mechanism of Kubernetes. (ii) We present InfraComposer, a policy-driven middleware that automates smart and adaptive workflow deployment in the cloud, leveraging domain-specific knowledge about the tools involved in the workflows. It further re-composes the deployment plans based on the scaling policies and the execution history towards more efficient and faster executions. And lastly (iii) we present DataBlinder, a distributed data-access middleware that encapsulates the complexity of the protection tactics to enable search and computing on encrypted data. It enables software service providers to securely and in a configurable manner outsource sensitive as well as non-sensitive data to the cloud. The middleware architecture is extensible to allow security experts to incorporate new tactics as well as provide security policies, and enables the developers to select the required protection level through certain abstractions.

The contributions are based on several application cases in the domains of aeronautics, finance, and healthcare. We have validated and evaluated our results. This thesis shows how they effectively address the above-mentioned challenges. Lastly, the dissertation outlines future directions to go beyond the limitations of the presented achievements.

Beknopte samenvatting

Software service providers passen steeds vaker het cloud computing-paradigma toe omdat het on-demand toegang biedt tot een virtueel onbeperkte bron van computer resources. Dit houdt doorgaans in dat ze hun programma's en gegevens uitbesteden aan cloud providers. Een gevolg hiervan is dat al de software services, gevoelige bedrijfsgegevens en processen van de software service provider zich in de cloud omgeving bevinden. Twee van de belangrijkste uitdagingen waarmee de industrie geconfronteerd wordt, zijn (i) hoe de uitbestede rekenkracht op efficiënte en eenvoudige wijze kan ingezet worden voor flexibele en geoptimaliseerde deployments en (ii) hoe de uitbestede opslag veilig kan gebruikt worden in een niet-vertrouwde omgeving, zoals bijvoorbeeld de publieke cloud, en toch op pragmatische wijze de functionaliteit van deze gegevens behouden kan worden. Om de implementaties te optimaliseren, hebben cloud providers hun computerinfrastructuur verbeterd, b.v. door de introductie van frameworks voor container orkestratie om zo de automatisering en flexibiliteit van software-implementaties te verbeteren. Bovendien migreren software service providers op steeds hogere software niveaus hun complexe en rekenintensieve systemen, zoals engineering workflows, naar de cloud om zo de efficiëntie te verhogen. Om het nut van beschermde gegevens in de cloud te behouden, hebben onderzoekers een brede waaier aan cryptografische technieken voorgesteld voor het uitvoeren van zoekopdrachten en berekeningen in versleutelde gegevens.

Software ontwikkelaars worden echter geconfronteerd met verschillende uitdagingen. Eerst, het elastisch schalen van computer resources, zoals containers, veroorzaakt niet te verwaarlozen vertragingen, ook bekend als cold start, deze vertragingen hinderen een flexibele en snelle grootschalige implementatie van software services. Dit probleem stelt zich vaak bij serverless computing, en meer algemeen bij systemen die automatisch schalen, voor toepassingen met op deadlines gebaseerde service-level objectives (SLO's). Er zijn veel technieken voor het instantiëren van resources met lage latentie in verschillende lagen van de infrastructuur. De bestaande oplossingen voor cold start in Kubernetes, de de facto standaard in container orkestratie, zijn echter nog steeds niet in

staat om het cold start probleem volledig te verhelpen. Ten tweede vereisen gedistribueerde en rekenintensieve applicaties, met name engineering workflows, op maat gemaakte en reproduceerbare implementaties om de uitvoeringstijd in de cloud te verkorten. De uitvoering van deze klasse van applicaties kan dagen duren, maar de juiste samenstelling van cloud resources kan de doorlooptijd mogelijks tot minuten terugbrengen. Het is echter niet triviaal voor technische domeinexperts om vooraf de hoeveelheid en het type van de vereiste cloud resources te bepalen en kosteneffectieve beslissingen te nemen voor toekomstige iteraties van hun implementaties. Ten derde zijn de cryptografische technieken voor zoeken en rekenen op versleutelde gegevens in de cloud divers in termen van beveiliging, prestaties en expressiviteit van zoekopdrachten. Bovendien zijn de onderliggende concepten en implementatie details van deze schema's vaak ingewikkeld voor applicatieontwikkelaars. Naast het foutgevoelige proces voor het selecteren van een schema, is de integratie van deze gegevensbeschermingstechnieken in heterogene en polyglot software, b.v. microservice-architecturen, een niet-triviale taak voor beveiligingsexperts. Ten slotte is de mogelijkheid om cryptografische schema's in deze omgeving in te pluggen een belangrijk aspect, aangezien er regelmatig nieuwe cryptografische constructies met nieuwe eigenschappen worden ontwikkeld. Deze problemen vereisen een zekere mate van cryptografische agility.

Het doel van dit proefschrift is om de bovengenoemde uitdagingen aan te pakken, om zo het proces van het uitbesteden van berekeningen naar de cloud te vergemakkelijken via efficiënte implementaties, en om gegevens pragmatisch op te slaan in de cloud met behulp van geavanceerde beveiligingen. Om dit te bereiken, zijn onze bijdragen drieledig: (i) we presenteren en evalueren drie benaderingen om de latentie bij cold start tijdens elastische schaling van containers in Kubernetes te verminderen. Om dit te bereiken, presenteren we verschillende technieken, zoals het delen van software bibliotheken, het proactiev aanmaken van netwerkbronnen en het toepassen van imperatief configuratiebeheer, dit laatste is in tegenstelling tot het bestaande declaratieve configuratie mechanisme van Kubernetes. (ii) We presenteren Infra Composer, een beleidsgestuurde middleware die slimme en adaptieve workflow-implementatie in de cloud automatiseert, gebruikmakend van domeinspecifieke kennis over de tools die bij de workflows betrokken zijn. Het stelt verder de implementatieplannen opnieuw samen op basis van het schaalingsbeleid en de uitvoeringsgeschiedenis van de toepassing om zo efficiëntere en snellere uitvoeringen te bekomen. Tot slot (iii) presenteren we DataBlinder, een gedistribueerde middleware voor gegevenstoegang die de complexiteit van de beveiligingstactieken inkapselt om zoeken en berekeningen op versleutelde gegevens mogelijk te maken. Het stelt software service providers in staat om op een veilige en configureerbare manier gevoelige en niet-gevoelige gegevens uit te besteden aan de cloud. De middleware-architectuur is uitbreidbaar om beveiligingsexperts in staat te stellen

nieuwe tactieken op te nemen en beveiligingsbeleid te bieden, daarnaast stelt DataBlinder de ontwikkelaars in staat om het vereiste beveiligingsniveau te selecteren door middel van bepaalde abstracties.

De bijdragen zijn gebaseerd op verschillende toepassingsgebieden zoals luchtvaart, financiën en gezondheidszorg. We hebben onze resultaten gevalideerd en geëvalueerd. Dit proefschrift laat zien hoe deze contributies de bovengenoemde uitdagingen effectief aanpakken. Ten slotte schetst het proefschrift mogelijke richtingen om verder te gaan dan de beperkingen van de gepresenteerde bijdragen.

List of Abbreviations

AOT Ahead Of Time. 110

ASLR Address Space Layout Randomization. 112

AWS Amazon Web Services. 6

CAD Computer-aided design. 9

CAE Computer-aided engineering. 9

DET Deterministic Encryption. 8

DHT Distributed Hash Table. 117

FaaS Function as a Service. 1, 4, 5, 10, 11, 107, 108, 110, 111

FHE Fully Homomorphic Encryption. 8

HE Homomorphic Encryption. 8

HIPAA Health Insurance Portability and Accountability Act. 6

HPC High Performance Computing. 3

HSM Hardware Security Module. 6, 115

IaaS Infrastructure as a Service. 1, 3, 4, 107

JIT Just In Time compilation. 110

JVM Java Virtual Machine. 110

NIST National Institute of Standards and Technology. 114

OPE Order-preserving Encryption. 8

ORE Order-revealing Encryption. 8

OS Operating System. 107

PaaS Platform as a Service. 3, 4

PPE Property-preserving Encryption. 8

PUF Physical Unclonable Functions. 115

SaaS Software as a Service. 2, 4

SE Searchable Encryption. 8

SLA Service Level Agreement. 4, 113, 114

SLO Service Level Objective. 4, 10, 16, 114

SPI Service Provider Interface. 20

TDE Transparent Data Encryption. 7

TEE Trusted Execution Environment. 7, 115

TOSCA Topology and Orchestration Specification for Cloud Application. 4

TPM Trusted Platform Module. 115

UDF User Defined Function. 20

VPN Virtual Private Network. 6

Contents

Chapter 1

Introduction

In the last decades, our daily lives have become more dependent on software-intensive systems, from banking to healthcare. Most businesses rely on software solutions to optimise their services and processes to deliver higher levels of customer satisfaction. The rapid growth of the customer base and their expectations drive such businesses towards more effective and cost-efficient models of computing infrastructure. To benefit from on-demand access to nearly unlimited and diverse pools of resources, the most prominent model over the last decade has become *outsourcing data and computational resources* to the infrastructure of cloud providers [37]. However, this outsourcing model has brought along several technical and societal challenges for both the cloud and software service providers.

Cloud providers offer different service models, varying from Infrastructure-as-a-Service (IaaS) to Function-as-a-Service (FaaS). They continuously improve the underpinning infrastructure to be more efficient, agile, secure and robust by employing and optimising various virtualisation technologies. For instance, virtual machines are the de facto, industry-standard infrastructural components offered by cloud providers. However, recent advancements in execution models of cloud-based software systems, such as serverless computing [10], require more agile and reproducible setups. As a result, operating-system-level virtualisation, such as containers, has gained popularity because of their faster startup time and their potential to act as relatively lightweight hosting environments [181]. These advancements and the ever-increasing demand for more efficient infrastructure have triggered cloud providers to optimize the underpinning systems rigorously.

The cloud computing paradigm has improved the business agility of software service providers by offering on-demand services enabling resources to be added

or removed instantly [37]; however, offering on demand services requires a high degree of resource configurability as a feature. For example, there are diverse types of virtual machines, varying from memory- to compute-optimised instance types. On top of that, a very large ecosystem of automation and orchestration tooling has emerged as a consequence of this flexibility. That leads to capacity-planning complexities [141, 106] during the initial steps as well as while software applications are running. For example, in aeronautics, it is not trivial for aerospace engineers to decide upfront the amount and type of required computational resources to perform an aircraft simulation and make cost-effective decisions for the future iterations of their experiments [133].

In the scope of outsourcing data to cloud providers, most businesses with mission-critical systems and sensitive data are concerned with regulatory compliances and data protection regulations [173]. Although cloud computing offerings unlock many economical opportunities, cloud providers are still considered to be honest in providing services, but at the same time potentially curious to learn more. In fact, in most cases, the data protection demand is requested by customers of software service providers. For example, a hospital that uses a Software-as-a-Service (SaaS) application to store medical data in the cloud, expects their data to be protected via encryption. To meet the privacy requirements, cryptographic key management should be handled by the software provider, outside the cloud storage infrastructure. This setting introduces new technical challenges if software providers aim at performing certain operations such as search and computation without decrypting the data stored in the cloud, or transferring the entire database to the client. To tackle this problem, the cryptography research community has developed a large repertoire of pragmatic protection techniques [71, 27, 1]. However, putting them into practice is not trivial in existing cloud-based software systems for both the security experts and software developers [8, 146, 2]. This situation screams for new data access systems with cryptographic agility, meaning that the extension and usage of techniques should be made easy and less error-prone.

This dissertation focuses on addressing the challenges of outsourcing data and computation to public cloud infrastructure through adaptive and reflective middleware platforms. In particular, we improve support for such computational model through (i) a middleware to compose and auto-scale execution infrastructure for the deployment of real-world engineering applications, (ii) an approach to improve the auto-scaling speed of computing instances, and (iii) a data access middleware with an extensible and cryptographically agile architecture to enable search and computation on encrypted data.

This chapter first presents the context of outsourcing computation and data to the cloud, including the motivation and the scope of this thesis. Second, the chapter outlines some key challenges and requirements with regard to several

application scenarios in the context of the aeronautics, automotive, FinTech and healthcare industries. Next, the goals and research approach of this dissertation are discussed. Subsequently, it presents the three major contributions of this work. Finally, this chapter concludes with an overview of the structure of the rest of this dissertation.

1.1 Outsourcing computation to the cloud

Major industries, such as aerospace and automotive, employ various sets of orchestrated, specialised software to simulate and optimise complex designs. Engineers use different hardware to deploy and execute these workflows, e.g., their desktop computers or High-Performance Computing (HPC) clusters. Because of capacity, resource configurability and agility reasons, cloud computing is a prominent option for outsourcing the computational infrastructure to cloud providers. In this subsection, we present various concepts and the components required for flexible and efficient cloud-based deployments. The subsection starts with the configurability of cloud platforms, and it further presents high-level background regarding autoscaling and virtualisation optimisation.

Configurability of cloud computing platforms. NIST defines cloud computing as a model for enabling on-demand network access to (theoretically) an unlimited pool of configurable resources such as networks, servers, storage, and services [140]. These resources can be quickly provisioned or dismissed by cloud users. For instance, OpenStack [157] is an open standard cloud computing platform developed by Rackspace and NASA. An outsourced application to this platform can be deployed using various types of computing instances, networks, storage types, images, and so on. The configurable options can be even more fine-grained depending on the complexity of the platform. For example, the computing instances can be either classic virtual machines or containers, different flavours with various numbers of cores and memory capacity, or even compute-optimised instances equipped with processor affinity (CPU pinning).

Infrastructure composability and orchestration tools. To run a complex software system on the cloud, for example an engineering workflow to optimise and simulate the electrical wiring of an aircraft cockpit, cloud orchestration helps with the automation of the deployment process using various IaaS and Platform-as-a-Service (PaaS) capabilities. To compose a software-defined infrastructure for such applications, the aforementioned configurable cloud components should be modularised in a way that can be flexibly combined. This enables

businesses to streamline their resource allocation, and workload distribution of software applications. For example, the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard language to describe the topologies of cloud-based services, and their underpinning components, relationships and the process that manages them. Using a realization of this standard, e.g. Cloudify, complex applications can compose their services and computing infrastructure as agile as possible without vendor lock-in concerns.

Elasticity and auto-scaling of resources. Elasticity in the context of cloud computing means the degree in which a software system is able to adapt to workload changes by adding or removing computational resources with the goal of meeting service-level objectives (SLO) [93, 92]. These resources can be re-scaled vertically or horizontally. Vertical scaling changes computational capacity of a computing component, such as adding more CPU power to a virtual machine. Horizontal scaling rescales the computing components by adding or removing extra replicas, such as adding more virtual machines.

Auto-scaling and serverless computing. In the context of self-adaptive systems, software services should adapt their computational capacity in case of unexpected workload changes. This concern is not limited to SaaS applications developed with IaaS and PaaS service-execution models. Recent models such as serverless computing, e.g. FaaS, also require elasticity of the underpinning computational components. To achieve cost-effective elastic scaling, in particular based on SLOs, two main classes of approaches have received remarkable attention by the researchers and practitioners, namely (1) reactive auto-scaling systems that define threshold-based scaling rules and policies, and (2) proactive auto-scaling systems [49] by employing predictive mechanisms such as reinforcement learning.

Rapid elastic scaling through virtualisation layer optimisation. Next to auto-scaling techniques, to meet the SLA requirements, auto-scaling systems should be able to launch computational resources as quickly as possible. To achieve this, cloud and service providers aim to optimise and improve the underlying computational infrastructure. In particular, the virtualisation layer has gained considerable traction. The classic virtualisation layer virtualises computer hardware, storage mechanisms and computer networks. In this scope, for example, hardware-assisted virtualisation improves performance accelerations with the help of hardware capabilities (e.g. CPUs). However, isolating applications with VM's causes portability issues and needlessly duplicates kernel resources. Operating-system-level virtualization, such as Linux Containers,

try to alleviate these problems by sharing the operating system, using kernel isolation mechanisms to separate the applications from each other. Containers take advantage of Linux `cgroups` and namespace isolation to set limits on resources and enable complete isolation for applications (e.g. process trees, networking, file system, etc.). In addition to the benefits such as runtime isolation, cost-effectiveness and portability, containers considerably improve the deployment speed of software services. This has a direct impact on rapid elasticity of resources as well as the speed of launching functions in the FaaS model.

1.2 Outsourcing sensitive data to the cloud

Businesses, such as healthcare and FinTech, with sensitive data are cautious about outsourcing their customer data to the cloud-based storage services despite the cloud providers' promises of high degree of security. However, to benefit from cloud economics, these businesses require to tackle a number of challenges. In this subsection, we present the context including the risks and the state-of-the-art approaches for outsourcing data to untrusted environments, such as public cloud platforms, in a pragmatic and protected fashion. In particular, we present high-level background concerning various threat models, legal requirements, the state of practice, and the advanced encryption techniques for protecting data in use.

Real-world threats of outsourcing data to the cloud providers. Generally, there are several types of threats and adversaries against outsourced data in the scope of public cloud platforms. The first type of adversaries are the tenants deployed next to the applications with sensitive data. Typically cloud providers colocate their customers on the same compute nodes as much as possible to maximise resource utilisation due to cost-efficiency reasons. This opens up new opportunities for malicious tenants to access restricted data by abusing known and unknown vulnerabilities in the underlying shared layers. For example, critical vulnerabilities such as Meltdown [125] and Spectre [112] exploit speculative execution in modern processors to render all protections useless and steal sensitive data from the memory of other running programs. This line of attacks are not limited to general-purpose processors. In fact, applications using confidential computing, in particular Intel SGX, are proven to be vulnerable as Foreshadow attacks [35, 201] managed to leak secrets protected within SGX enclaves. This affects virtual machines, hypervisors, operating system kernel memory and so forth.

Next to the adversaries who aim to actively and maliciously steal secrets, the second type of threat is based on adversaries who have copies of sensitive data at various times without the query logs. This threat is called a snapshot adversary and the malicious act is widely known as *a data breach*. In a public cloud setting, a data breach can happen by storage configuration mistakes, insecure APIs, and cloud providers' insider threats such as malicious system administrators. The threat is not limited to the breach of plaintext data. Recent research [148] showed that even the breach of encrypted databases with medical data can lead to plaintext recovery of a considerable amount of real patient data. In this track, researchers only took advantage of techniques such as frequency analysis, sorting and combinatorial optimisation.

Lastly, in most threat models, cloud providers are considered to be honest but curious. In addition to the financial, political or insider-threat motivations for looking into cloud-based storage systems, these companies are bound to local jurisdictions. For example, the Foreign Intelligence Surveillance Amendments Act (2008) of the United States, in particular Code § 1881a, permits coercion of cloud providers in secret mass-surveillance from within their data centres.

The need for data protection by law. To keep the customers of software service providers, e.g. patients of hospitals, protected against the aforementioned threats, there are various data protection regulations such as GDPR in the EU. For instance, the Health Insurance Portability and Accountability Act (HIPAA) of the United States defines a breach notification rule (45 CFR §§ 164.400-414). It demands that healthcare businesses to notify the patients in case of any breach of protected health information. Moreover, it recommends the use of appropriate encryption techniques for such data.

State-of-practice data protection mechanisms. There are three major states of digital data that require confidentiality, integrity and availability [152]:

- *Data in transit.* Data in transit refers to when data flows over protected or unprotected networks. Protection mechanisms in this scope secure the client-server interactions such as SSL/TLS in browsers, DTLS in datagram-based applications, and IPsec in virtual private networks (VPN).
- *Data at rest.* This state is related to protecting data that is physically stored in storage systems, for example by using standard encryption AES-256 with key materials managed by hardware security modules (HSM). Cloud providers offer key management services to enable data protection at rest for storage services. For example, Amazon Web Services (AWS) provides a service called KMS to manage cryptographic keys by using HSM

under the hood for securing services through data encryption. Likewise, Microsoft Azure offers a service called Key Vault for the same purpose.

- *Data in use.* This state refers to active data that is stored in a non-persistent state (e.g. in RAM or CPU caches). In other words, a certain set of operations is supposed to be performed over this data. At the time of writing this thesis, to protect sensitive data in this state, cloud providers primarily offer hardware-oriented solutions based on trusted execution environments (TEE). For example, in Intel SGX, they use a concept called *enclave*, meaning that sensitive content are protected in a way that other processes outside of the enclave cannot access the content, including the operating systems and hypervisors. In the semiconductor industry, Intel SGX and ARM TrustZone are the well-known examples of TEE architectures. As a cloud offering, for example, Microsoft Azure offers confidential computing through SGX-enabled containers or virtual machines.

Data protection support in SQL and NoSQL databases. Next to firewalls in front of database servers, major databases, be it SQL or NoSQL, offer various protection levels. The mainstream relational databases support transparent data encryption (TDE). For example, MySQL, as one of the top-3 relational databases [58], offers data-at-rest encryption through encrypting tablespaces right before persisting data on the physical files. PostgreSQL, OracleDB and MS SQL Server also offer the same feature. In the scope of NoSQL databases, likewise, most databases support TDE. It is worth mentioning that TDE is mostly supported in the enterprise versions of these databases such as MySQL while many applications use the free and open-source variations.

The need for advanced pragmatic tactics to protect data in use. Client-side encryption is one of the prominent approaches to protect data. In other words, the aim is to perform data encryption at the client side, either in the datacenter of software providers or client applications, outside of the infrastructure of cloud providers. Therefore, cryptographic keys should never be transmitted to the untrusted environments. Among the SQL databases, PostgreSQL offers client-side encryption. However, to make practical systems, such a security architecture does not allow certain functionalities like search or computing on data in a PostgreSQL instance running in the cloud, because the client-side key management renders the server-side operations useless. This happens due to the absence of cryptographic keys and the lack of search indexes. Another example is the Amazon DynamoDB Client which ensures data protection in transit and at rest, but it falls short in complex cloud-side data operations.

In brief, businesses require protected outsourcing of sensitive data to untrusted environments, and at the same time they need to be able to perform search and computation over encrypted data.

Secure search over encrypted data. The cryptography community has been conducting research for more than two decades on pragmatic encryption constructions, generally known as searchable encryption (SE). These schemes enable search over encrypted data by producing cryptographically secure search indexes. These schemes generally enable cloud providers to search for user-requested keywords on encrypted data without knowing the content of the queries and the plaintext data. The secure index typically reveals no information (or formally defined leakage) about the content of search words and the data itself. These schemes are quite diverse in terms of security, performance and query expressiveness. For example, there exist symmetric SE schemes that support equality search as well as boolean queries. More practical, albeit less protective mechanisms, are based on property-preserving encryption (PPE), such as deterministic encryption (DET), order-preserving encryption (OPE) and order-revealing encryption (ORE) schemes. In brief, each scheme guarantees a certain level of security and offers specific functionality at a particular performance cost.

Computing on encrypted data. To perform different aggregation queries, such as a request to calculate average heart rates or body mass index (BMI) over a population within a dataset, the computation should be done within, or as close as possible to the database. However, to satisfy the data protection requirements, cryptographic keys should not be transmitted to the cloud provider infrastructure. Researchers and practitioners developed advanced encryption primitives to achieve this goal by introducing homomorphic encryption (HE). The HE schemes encrypt data in a way that their underpinning mathematical properties enable the applications deployed in the cloud to perform certain operations directly over encrypted data such as addition or multiplication. There are different flavours of such schemes. For example, some schemes only offer the addition functionality such as Paillier [159], and others only multiplication like ElGamal [64]. Somewhat HE (SHE) and Fully HE (FHE) offer some combination of both at the cost of performance (e.g., BGV [31] and TFHE [52]).

In a rapidly-changing computing infrastructure landscape, businesses face several challenges in the context of cost-efficient and secure outsourcing that will require further research. The next sub-section outline the challenges that this dissertation investigates to address.

1.3 Outsourcing computation and data: challenges

In this dissertation, we take an application-driven approach to identify the key challenges for outsourcing computation and data to the cloud. We therefore present five real-world application cases (see Table 1.1), three in the context of outsourcing computation (A1-3) and two in the context of outsourcing data (A4-5) to the cloud.

Challenges for outsourcing computation. In aeronautics and automotive, engineers use software-based workflows for simulation and optimisation of complex designs. The first two application scenarios (A1 and A2) are related to such systems. These types of engineering workflows are executed by special engines, e.g. Optimus [185]. These engines typically orchestrate a task given some input values for a chain of engineering applications. For example, such workflow engines are capable of running multi-disciplinary design optimisation techniques by automating computer-aided design (CAD), computer-aided engineering (CAE) and proprietary software. There are a number of challenges to execute these workflows on the cloud.

C1 *Long-running execution:* The execution of these workflows typically takes minutes to days. These workflows are not deadline constrained; however, for example, a quasi-exhaustive search in a large set of possibilities for optimising the hinge system of an aircraft rudder should run faster. This allows the engineers to produce solid optimisation and simulation results by performing more experiments. This is achievable by improving the underlying computing infrastructural components.

C2 *Repetitive deployment and iterative execution:* These optimisation experiments are subject to retuning and redeployment. For example, if the workflow outcome is not desirable, engineers typically re-run the workflow with different input data or with other optimisation techniques. This often occurs at different points in time, and as a result, for economical reasons, the cloud components should be decommissioned and instantiated frequently. For engineers without expertise in cloud operations, the management of such large-scale setups is a complex and cumbersome process.

C3 *Non-trivial and efficient composition of resources:* Selecting the right types and amount of resources for computation, storage and networking is not a straightforward task. For example, collocating compute-intensive engineering tools on the same VM hinders the performance, which results in the slow overall execution of the workflow. And most importantly, the

	Application case	Challenge scope
A1	Aircraft rudder simulation and optimization	Composing optimal cloud resources [C1, C2, C3]
A2	Electrical wire harness simulation	Composing optimal cloud resources [C1, C2, C3]
A3	Job processing microservice and FaaS	Slow and inefficient container bootstrapping [C4]
A4	Electronic health record software	Development and integration complexity of protection techniques regarding search and computing on encrypted data [C5, C6, C7, C8]
A5	Invoice management software	Development and integration complexity of protection techniques regarding search and computing on encrypted data [C5, C6, C7, C8]

Table 1.1: Overview of the challenges faced in outsourcing computation and data to the cloud providers stemming from various real-world application cases. A3 is an application deployment pattern used in cloud-native application cases and not particularly an application case on its own.

execution behavior of these domain-specific tools is heavily dependent on the workflows and their input parameters. The main objective is to increase the speed of such experiments such that engineers are empowered to rapidly perform iterative runs of their experiments.

Therefore, a smarter, history-driven and controllable approach is required to automate the cloud infrastructure in a software-defined fashion.

To further improve the efficiency of the cloud-based software systems, computing components should be instantiated as quickly as possible. For example, in the third application case (A3), the job producers place tasks on a queue, and based on the queue load and the required delivery deadline, a number of workers are instantiated in order to process the tasks. The delay caused by the time to bootstrap computing components has an impact on applications with deadline-based Service-Level Objectives (SLO). In the function as a service (FaaS) architecture, likewise, when a function is called, an instance of the function should be launched rapidly to mitigate SLO violations. As mentioned

in the previous subsection, practitioners use lightweight OS-level virtualisation techniques such as Docker containers to tackle this problem; however, this is still not sufficient for most enterprise applications, especially for the FaaS case.

C4 *Cold start problem:* Based on different workloads, upon each function call or elastic scaling of workers, there is a latency which is known as cold start. The cold start problem is caused by the time to (1) bootstrap containers, (2) prepare the software environments, and (3) initialise the user code. And, it has a considerable impact on most applications [33, 107]. Related work has introduced various techniques to reduce this latency by using snapshots, lazy fetching of Docker images, and container queues. However, most techniques are concerned with different tradeoffs. For example, queue-based approaches pre-launch various pools of containers in advance to reduce the cold start latency at the cost of excessive memory duplication. Therefore, further optimising the infrastructure, in particular using the industry-driven de-facto container orchestration frameworks like Kubernetes, is still an open problem.

These challenges ($C1$–4) have emerged from the aforementioned application cases ($A1$–3), covering two large classes of software systems that the industry has been aiming to host and run on the cloud.

Challenges for outsourcing data. Healthcare providers such as hospitals use cloud-based software to store, retrieve and query patient records and other medical data (application case A4). This type of software is often offered by third-party software service providers. Similarly, financial organisations like banks use billing services to streamline their invoicing workflows (application case A5). Billing software service providers use the cloud for the generation, processing, and storage of documents. In both application cases, clients own sensitive data which is supposed to be processed, stored, and queried by service providers. These providers typically use cloud-based storage systems. As explained earlier in this chapter, cloud providers are not within the trust boundaries of such application scenarios. To protect sensitive data using advanced data encryption in a pragmatic way to preserve utility, various protection schemes have been presented by the research community. Integration of these techniques into real-world systems has introduced several challenges.

C5 *Diversity of protection schemes:* There is no one-size-fits-all cryptographic scheme that maximizes all three aspects of the security, performance, and functionality trade-off. Tactics enabling secure search, in particular searchable encryption, are quite diverse. For example, encrypting the

whole database (AES256) without searchability in mind provides us with a high degree of security but falls short of performance. Some tactics leak less information than others; among them, there are some with sub-linear search complexity. Each of these schemes enables different search functionalities (e.g. equality, conjunction, disjunction, etc.) and aggregate queries (e.g. sum and average). Therefore, it is not possible and trivial to choose a general-purpose tactic.

C6 *Complex implementation:* The underlying concepts and implementation details of these cryptographic constructions are complicated for application developers most of the time; in other words, choosing the right scheme as well as a secure and correct implementation is prone to mistakes. For example, developers might overlook side-channels in their implementations, or they use unauthenticated modes of symmetric encryption only because developers heedlessly copy code snippets from the web [68]. In the scope of the encrypted search, it is not hard to build searchable encrypted databases that are susceptible to almost full recovery attacks using frequency analysis [148]. It is shown that software developers without cryptography expertise most likely make wrong choices [24, 147]. Therefore, both the implementation of the cryptographic constructions and also the API usage are complex and error-prone tasks.

C7 *Complex integration and crypto agility:* Integrating data protection tactics in the form of libraries to heterogeneous and polyglot software, e.g., microservice architectures, is prone to mistakes because such systems are developed using various ecosystems of programming languages. Furthermore, developing and incorporating new cryptographic schemes in an existing software stack is not a trivial and straightforward task for security experts. In fact, in this scope, the ability to plug and play cryptographic schemes is an important aspect since new constructions with new properties are being presented frequently. Potentially, future techniques or bug fixes can unlock new security properties, performance improvements and functionalities. Therefore, rigid software architecture complicates the extensibility of software systems.

C8 *Viability of the missing functionalities in existing storage systems:* Most schemes, such as homomorphic encryption or searchable symmetric encryption, require extra functionalities to be available within the database or as close as possible to the data. For example, to perform additive aggregations homomorphically in the Paillier cryptosystem [159], ciphertext values are required to get multiplied. Next to that, to perform a search using a token against an encrypted index, existing database index techniques are useless. The research community has presented new storage systems such as CryptDB [166], Blind Seer [160], EncKV [207],

HardIDX [70], and so on. However, these systems are either custom or designed with rigid architecture without extensibility in mind. Therefore, no attention has been paid to cryptographic agility.

These challenges ($C5$–8) have emerged from our healthcare and FinTech application cases ($A4$ and $A5$) where software service providers aim to store sensitive data like medical records or customer invoices on cloud-based storage systems.

To summarise, outsourcing computation to the cloud is challenging for distributed applications such as engineering workflows due to the complexity of composing optimal cloud resources. Moreover, for cloud-native applications that require elastic scaling of computing resources, the cold start problem can cause SLA violations. Therefore, this class of applications require minimal bootstrapping time such as the case for application containers in the Kubernetes orchestration framework. Furthermore, outsourcing data in a protected fashion comes with several challenges in the scope of development and integration complexity of protection techniques for software developers and security experts.

1.4 Goals and research approach

This dissertation focuses on challenges that emerged from distributed software deployment efficiency and data security for the cloud in an effortless, efficient, reliable, secure and pragmatic way. Consequently, the research approach is composed of (1) studying existing research, (2) incepting and developing new techniques, and (3) evaluating and validating the approaches in industrial application cases. This dissertation advances the state of the art by introducing new methods and middleware platforms to enable compute-intensive software systems to run in the cloud. To achieve this, the overall research goal can be broken up into the following goals:

G1 *Infrastructure agility and efficiency.* We define cloud infrastructure agility as an attribute of a platform that offers customisability at different levels of granularity (e.g. software colocation, VMs, CPU, Memory, etc.). And most importantly, it should be customisable through orchestration software tools. Our key sub-goals are:

(a) *Automated and controllable capacity planning and deployment of engineering workflows on the cloud*: Infrastructural agility allows clients to perform horizontal scaling, vertical scaling, software co-location, dynamic storage and network allocation, and so forth. The

high degree of customisability typically results in usage complexity. We aim to abstract away this complexity and facilitate novice and expert application developers (in this case engineers) to employ *efficient* computing infrastructure for their applications at hand; furthermore, our end optimisation goal is to automate the entire process as much as possible.

(b) *Low-latency horizontal scaling of computing resources for applications with deadline-driven Service-Lavel Objectives (SLO).* Our goal is to have a better understanding of the industry-standard container orchestration frameworks as well as the existing cold start mitigation techniques. The end goal is to improve the startup time of application containers; this goal has an impact on applications with deadline-driven SLOs or serverless computing.

G2 *Abstractions for complex and distributed protection tactics.* Middleware systems typically aim to abstract away complex concepts, error-prone implementation, and configurations. This is mostly achieved by introducing certain abstractions for simplicity, usability, and configurability of those underpinning concepts. Our key sub-goals are:

(a) *Enabling data access middleware to search and compute over encrypted data.* Our main goal is to present new abstraction models for non-functional requirements such as data security. We aim to enable application developers to persist sensitive data on the cloud in an encrypted form and yet be able to perform certain operations such as search and aggregation.

(b) *Improving crypto agility.* The ability to plug and play cryptographic schemes depending on their evolution in time, or switching between protection techniques at run time is called crypto agility.

G3 *Reusability of the approaches.* The aforementioned challenges can be addressed in a domain-specific approach with tailor-made architectures. However, our goal is to present reusable designs based on middleware-oriented approaches, through which application developers, domain experts in general, would be able to take advantage of our contributions in a reusable and modular way.

To achieve these goals in this dissertation, we investigate the feasibility and applicability of adaptive and reflective middleware and infrastructure architecture. Adaptation is achieved through reconfigurable architecture, and the key property for reconfigurability of an adaptive architecture is openness. Typically, not all aspects of a system are realised to be configurable. The

openness and configurability of the architecture are feasible through reflecting these aspects via meta interfaces.

In the scope of goal $G1$, an adaptive middleware architecture enables software systems to adapt their structure, behavior, and resources based on certain conditions. For example, a software system that runs an engineering workflow in the cloud does not reflect certain non-functional aspects such as resources and deployment components. In this context, to spin up efficient and tailor-made infrastructure for such workflows, these concepts that inherently exist in the underlying layers of the systems should be reflected through middleware platforms. Therefore, they should become first-class citizens in our systems through reification. Our goal is to present new reflection meta models and reuse existing ones at the infrastructural level. This effort enables reflection at different layers from workflow components to cloud resources. In goal $G2$, we investigate a reflective architecture for the data access middleware, through which the behavior of the middleware itself can be reconfigured at runtime.

Research approach. This dissertation takes an application-driven approach to address the challenges listed in Section 1.3. In high level, the main idea is to abstract away complex concepts and functionalities in a reusable layer that is applicable to a large class of application cases. To achieve this, this thesis focuses on middleware solutions with the aforementioned research goals in mind. The research approach can be boiled down to the following steps.

1. *Driven by industrial application cases.* Our approach targets real-world applications from various domains of industry. The first step is to perform requirement analysis to understand the challenges faced by different stakeholders such as business owners, software developers, users and customers of these applications. The application cases used in this thesis are listed in Section 1.3 and are primarily based on a European and an ICON research project, namely ITEA3 IDEaliSM [104] (application cases A1 and A2 for outsourcing computation) and SeClosed [100] (application cases A4 and A5 for outsourcing data).

2. *Leveraging state of the art and proven technologies.* In this thesis, we leverage the existing research and state-of-practice technologies to advance the state of the art in cloud computing and data access middleware. In particular, we used and contributed to the de-facto cloud computing infrastructure such as OpenStack and Kubernetes. We developed new ideas built upon the seminal contributions of the research community such as searchable encryption, homomorphic encryption, reflective middleware, and various cold start mitigation techniques (application case A3).

3. *Validated by development of prototypes.* To validate the new approaches and the goals of this thesis, we developed software prototypes in collaboration with the industry partners. Our goal was to investigate the feasibility of the contributions and understand the advantages and shortcomings of the contributions. One of the main focuses of our implementations was on the reusability aspect to meet various goals including G3 regarding reusability.

4. *Evaluated extensively.* Each software artifact has gone through functional validation and extensive evaluation to measure the impact and effectiveness of the claims in terms of performance. The performance evaluations were composed of various series of experiments based on evaluating the techniques on a standalone basis, based on application cases, or measuring the overhead.

1.5 Contributions

The key contributions of this thesis are summarised in Table 1.2. Fig. 1.1 illustrates the overview of the contributions. The general goal is to facilitate the outsourcing of the data and computation for various applications, that require pragmatic data protection and deployment efficiency, to the cloud through our middleware framework (depicted in grey).

Contribution 1. Reducing cold starts during elastic scaling of containers in Kubernetes. In this contribution, we investigate the existing and new techniques for improving the agility of modern computing infrastructure for cloud-native applications, namely microservices with deadline-based SLOs. More precisely, the state of the art presents several techniques to mitigate the cold start problem. In the context of Kubernetes, the de-facto container orchestration framework in industry, we thoroughly investigate the impact of two of these approaches: *layered-based library sharing and pools of reusable network containers*. Kubernetes operates based on declarative configuration management that changes the state of the system through various control loops. We present an imperative approach to improve the speed, and combine it with the two cold-start mitigation techniques. The key findings of our investigation are:

- The layered-based library sharing approach results in a large reduction in the startup time of application containers.

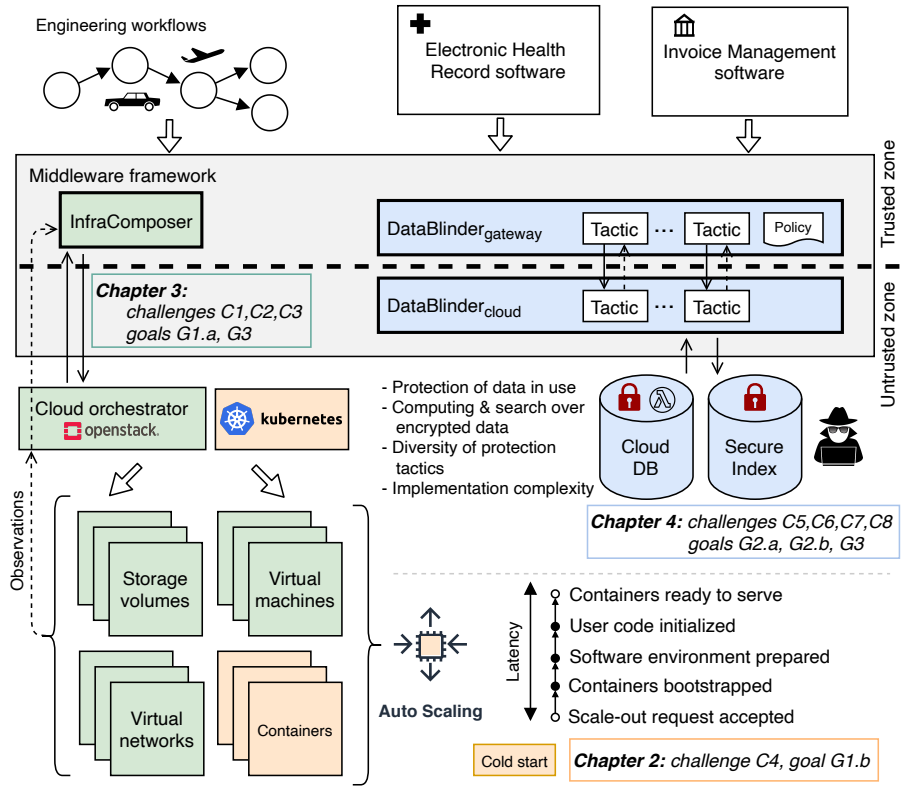


Figure 1.1: Overview of the contributions, goals, and challenges of this thesis. All software components situated below the dashed line are within the cloud provider's infrastructure. To outsource the computation of the engineering workflows, the *InfraComposer* middleware is contacted (see Chapter 3). The middleware composes and deploys the required cloud resources as a deployment plan with the help of a cloud orchestrator. The underpinning system and workflow resources are monitored, and the monitoring data is fed back to the middleware for future deployment optimisations. In Chapter 2, depicted in light orange colour (yellow), we combine three techniques to address the cold-start problem in Kubernetes. The cold-start latency affects the agility of auto-scaling systems. In Chapter 4, to enable applications that work with sensitive data, DataBlinder gateway offers regular database operations to the software developers, and under the hood, it incorporates distributed protection tactics for computing and searching over encrypted data.

- Pre-creating network containers has a greater impact when multiple application containers are started in parallel.

Contribution	Challenges	Goals
Cold start mitigation techniques (*Chapter 2*)	$C4$ Cold start problem	$G1.b$ Low-latency horizontal scaling of computing resources for applications with deadline-driven SLOs
InfraComposer middleware (*Chapter 3*)	$C1$ Long-running execution $C2$ Repetitive deployment & iterative execution $C3$ Non-trivial & efficient composition of resources	$G1.a$ Automated and controllable capacity planning and deployment of engineering workflows on the cloud $G3$ Reusability
DataBlinder middleware (*Chapter 4*)	$C5$ Diversity of protection schemes $C6$ Complex implementation $C7$ Complex integration & crypto agility $C8$ Missing functionalities in existing storage systems	$G2.a$ Enabling data access middleware to search and compute over encrypted data $G2.b$ Improving crypto agility $G3$ Reusability

Table 1.2: Overview of the challenges and goals per each contribution

- The imperative configuration management introduces startup time determinism and predictability, which makes the computing infrastructure more reliable for SLA-driven applications.

Contribution 2. InfraComposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation and optimization workflows. The second contribution of this dissertation is InfraComposer, a reflective and adaptive middleware that enables and manages smart, adaptive workflow deployment, scaling, and execution in the cloud. It streamlines the workflow

deployment in the cloud using a step-wise process by leveraging domain-specific knowledge about the tools. More concretely:

1. The middleware presents a deployment planning approach based on the annotation-driven resource reservation. Using these annotations, provided by engineering domain experts, an initial deployment plan is generated and deployed.
2. Through performing deep inspection of running tools, InfraComposer takes a history-driven approach to reconfigure the deployment plans adaptively based on the execution history.

The adaptive scaling and reconfiguration are driven by policies that can reason about the execution history. We therefore form a MAPE-K loop [110], which stands for monitoring, analysis, planning, execution and knowledge. To realise this process, we also present a reflective architecture with meta-models aiming to reify the key architectural, execution and resource utilisation concepts. These contributions are validated and evaluated with industrial use cases and scenarios, primarily in automotive and aeronautics in the frame of the European project ITEA3 IDEaliSM [104].

Contribution 3. DataBlinder: A distributed data protection middleware supporting search and computation on encrypted data. The third contribution of this thesis is DataBlinder, a distributed data-access middleware that encapsulates the complexity of the protection tactics. It enables software service providers to securely and in a configurable manner outsource sensitive as well as non-sensitive data to the cloud. The core contributions are as follows:

- The middleware enables the adaptive selection of data protection tactics for software developers through an abstraction model. This model enables software developers to request their desired protection level and types of queries at the field-level granularity. This approach can be implemented by annotating the database schema and potentially persistence entities demarcation. In the tactic selection phase, the right implementation is loaded at runtime based on the security policies.
- The architecture of the middleware is extensible and pluggable, in the sense that it is possible to introduce new protection tactics by security experts without cumbersome integration efforts. To achieve this, we present an abstraction model that reifies leakage profiles, performance metrics, and operations. The architecture is based on the service provider interface

(SPI) pattern [176], and we employed it in a distributed architecture with both a trusted and untrusted zone. We prototyped several searchable and homomorphic encryption schemes as a validation of the system.

- We investigated the possibility of dynamic deployment of User Defined Functions (UDFs) in major NoSQL databases. The main idea is to extend the existing database functionalities without any modification to the source code of databases, e.g., to support server-side homomorphic operations. We developed two prototypes for MongoDB and Cassandra to validate that the idea is practically feasible. However, our evaluations show that cryptographic UDFs, that run within database engines, do not guarantee the best performance for all NoSQL databases. We performed an analysis regarding this contribution in this dissertation.

Our prototype is implemented as a data-access gateway in a distributed micro-service architecture. Next to the overhead of cryptographic schemes, DataBlinder causes only 1.4% overall throughput loss. These three contributions are crucial avenues for research to foster crypto agility in the domain of outsourcing data to untrusted environments from the software engineering and middleware perspective. These contributions are validated and evaluated with industrial use cases in healthcare and FinTech in the frame of the ICON project SeClosed [100]. Our approach has been reviewed and validated by the AWS Cryptography team.

1.6 Outline of the thesis

The remainder of this dissertation is structured as follows.

Chapter 2 presents and evaluates the first contribution. In this chapter, we present the cold start problem and a thorough evaluation of three techniques for bootstrapping the computing infrastructure. This work has been published in the 36th ACM/SIGAPP Symposium On Applied Computing (SAC 2021) [94].

Chapter 3 presents, validates and evaluates the second contribution. In this chapter, we present a policy-based, history-driven, and adaptive middleware for the deployment of engineering workflows. This work has been published in the Journal of Systems Architecture (JSA) [16], as the extension of two prior publications in the proceedings of the 16th Workshop on Adaptive and Reflective Middleware (ARM 2017) [15], and the NAFEMS European Conference (2018) [133].

Chapter 4 presents, validates and evaluates the third contribution. In this chapter, we present a data-access middleware for secure outsourcing of data to the cloud platforms. This chapter is primarily based on our publication in the proceedings of the 20th ACM/IFIP/USENIX International Middleware Conference Industrial Track (Middleware Industry 2019) [17]. This chapter further evaluates the impact of dynamic deployment of user-defined functions. The author of this dissertation designed, prototyped, and performed an analysis as a co-author in a paper that has been published in the Journal of Information Systems [168].

Chapter 5 concludes this dissertation by summarising the contributions. This chapter further discusses the limitations of the contributions and outlines the future research avenues in this context.

Chapter 2

Reducing cold starts during elastic scaling of containers in Kubernetes

This chapter presents three techniques in the context of Kubernetes aiming at reducing *cold starts* during elastic scaling of containers. The delay caused by the time to bootstrap a container, often known as cold start, has an impact on applications with deadline-based Service-Level Objectives (SLOs). Our research combines and evaluates these techniques: (i) pre-creation of network containers, (ii) using container images that enable sharing of linked libraries in memory and (iii) extending the declarative configuration management approach of Kubernetes with imperative configuration for creating multiple application containers in parallel. A prototype of the approach is implemented and tested on a Java-based Spring Boot application where the cold start problem occurs due to various library dependencies.

Our findings illustrate that the use of containers that allow for library sharing already has a large, positive impact when starting up a single container. The pre-creation of network containers in combination with imperative configuration enables the application to meet deadline-driven SLOs without a non-deterministic delay that appears in Kubernetes when multiple containers are created in parallel. We conclude that the use of container images that allow for library sharing is a must for all applications that require fast container start-ups in Kubernetes. Pre-creation of network containers when combined with imperative configuration also has a positive impact on SLO compliance

during elastic scaling of containers.

This chapter is based on a publication at the 36th ACM/SIGAPP Symposium On Applied Computing (SAC 2021) [94]. The remainder of this chapter is structured as follows. Section 2.1 introduces the cold start problem and provides an overview of the state of the art. Furthermore, it outlines our contributions and findings in the scope of Kubernetes. In Section 2.2, we present the required background, and in Section 2.3, we outline related work on containers and the cold start problem. Then, Section 2.4 discusses how reusable network containers, library sharing and imperative configuration can be realized in Kubernetes. Subsequently, Section 2.5 presents our evaluation methodology, experiments and the results. Finally, Section 2.6 concludes our research and summarise our findings.

2.1 Introduction

New cloud computing execution models such as serverless computing have become popular recently [11]. Software providers offer their services through microservices or purely serverless architecture. In that regard, one of the emerging and widely adopted paradigms of delivering software is the use of OS-level virtualization such as containers [184]. To facilitate managing an application as a distributed set of containers, container orchestration frameworks are used. The popular examples are Docker containers and Kubernetes [20]. Kubernetes has pioneered in declarative configuration management, where a control loop detects differences between a desired and actual system state, and a policy-rich scheduler that takes into account expressive placement constraints for placing containers on a cluster of worker nodes.

Automatic scaling of functions is an inherent feature of serverless computing [11, 107]. Upon each function call, one or more containers need to be started or elastically scaled out based on different workloads. Even though containers are considered to have faster startup times compared to traditional virtual machines, the latency caused by the time to (i) bootstrap containers, (ii) prepare the software environments, and (iii) initialize the user code has an impact on some applications [33, 107], especially multi-tenant services with strict service level agreements (SLAs). This problem is called *cold start*. It sometimes takes seconds or minutes to have a container and the application up and running.

To reduce this cold start latency, various techniques have been already introduced, e.g. using snapshots [39], lazy fetching of Docker images [54] and container queues [122, 144, 136]. However, there are always trade-offs. In particular, the queue-based approaches mostly sacrifice memory to obtain a

faster start-up time since containers are pre-launched. In the context of the serverless middleware OpenWhisk, Mohan et al. [144] improved this deficiency by pre-creating and caching a pool of reusable networking endpoints, namely network containers. When a function container is created, an existing network container gets bound to it. This approach results in 80% reduction in execution time in comparison to cold queues and several orders of magnitude reduction in memory footprint. However, this technique has not yet been evaluated in the context of Kubernetes.

Moreover, in a serverless setting, or auto-scaling in general, software dependencies are redundantly loaded in memory when containers are replicated. In particular, recent research has shown that replicated containers can share common libraries in memory, provided that the used container image encapsulates the libraries in separate image layers. [67]. This technique has also been reported to reduce startup times as well [197].

Contributions. We extend Kubernetes with an imperative configuration management approach to reduce the cold start problem when auto-scaling containers in parallel.

We have found that in Kubernetes a non-deterministic delay, or more specifically a varying delay, appears in container start-up times when creating multiple application containers in parallel on the same node. The cause of this delay is due the declarative configuration management approach of Kubernetes where various controllers act upon differences between desired and actual system state.

It has already been shown before that declarative and imperative configuration management are complementary techniques [32]. Our imperative approach uses a script to create multiple containers in parallel. However, it builds upon the technique introduced by Mohan et al. [144] to still benefit from the advantages of Kubernetes' declarative configuration management approach and its policy-rich scheduler. More specifically, to realize a queue of reusable network containers in Kubernetes, we create a pool of empty pods (i.e. pod is a group of containers). Each pod comes with a network container since network containers can only be instantiated in a pod. To scale out the application in an imperative fashion, our scaler selects one or more of these pods and it launches a number of application containers by injecting these into the selected pods.

Our research goals are threefold:

1. The first goal is to see whether creating a pool of network containers (also called `pause` containers) in advance has a positive effect on the cold start problem in Kubernetes.
2. The second goal is to investigate the impact of container-based library

sharing on the cold start problem when combined with network containers technique in Kubernetes.

3. The last goal is to compare imperative and declarative configuration management along with the two aforementioned techniques. The imperative approach creates multiple containers by means of an imperative script, whereas the declarative approach employs a control loop with a policy-rich scheduler.

A prototype of the approach is implemented and tested on a Java-based Spring Boot application [187] where the cold start problem occurs. This application is a simple job processing microservice, which has a variable amount of users. The intention is to test the effectiveness of the approach with multiple users submitting jobs at the same time. As a validation, we compare our approach with using the default declarative approach of Kubernetes for increasing or decreasing the number of replicated Pods.

Findings. We have evaluated our approach with fluctuating workloads. We could deduce from the experiments that using container images that support library-sharing has the greatest impact on the "cold star" problem. Library sharing also had a positive but small effect on the start-up of multiple containers in parallel. After all, this provides an extra reduction in time for starting up application dependencies (e.g. the Java Virtual Machine (JVM) and Tomcat server). Finally, the effectiveness of the network container queue is limited, but in conjunction with the imperative approach, it results in faster boot time of containers since it can be done in parallel. The imperative approach further enables the application to meet SLA targets without a non-deterministic delay that appears in the declarative approach when multiple Pods must be created at the same time due to concurrent user requests.

2.2 Background

In this section, we introduce: (i) Kubernetes and some of its underpinning components, (ii) the differences between declarative and imperative configuration, and (iii) a brief definition of the cold start problem in the context of containers.

2.2.1 Kubernetes

Kubernetes is a container orchestration framework commonly deployed and used by researchers and practitioners. It facilitates the deployment, (auto)scaling and

management of container-based applications through declarative configuration files. A *pod* is a group of containers that share storage and network resources [113]. Pods are the smallest unit of deployment in Kubernetes. A *deployment* is used to get a pod or a *ReplicaSet* of a pod to a certain state. The *deployment controller* is responsible for changing the current state to the requested state. CPU and memory of compute resources (containers or pods) can be managed by guaranteed *resource requests* and *maximum resource limits*.

`Pause` containers. Each pod has a `pause` container. A `pause` container is responsible for the creation of a shared network and a namespace for the other containers in the pod [120]. If any container within a pod fails, the entire pod does not restart thanks to `pause` containers. This is because this container ensures that the network namespace and PID namespace remain. We use the terms `pause` and network container interchangeably throughout this chapter.

Internal components. In Kubernetes, the control plane, which manages the worker nodes and pods, is composed of schedulers, API servers, and an `etcd` database. Schedulers are responsible for suitable pod placements based on different scheduling decisions. API server exposes the Kubernetes functionalities. The `etcd` database is a consistent and highly-available key-value store as a backing store for all cluster data [113]. The worker nodes host pods. *Kubelet* is an agent on each node responsible for making sure that containers are running and healthy [113].

2.2.2 Declarative and imperative systems

Modern software systems and infrastructure can be configured and instantiated by code. In general, similar to two major programming paradigms, namely the declarative and imperative paradigms, the state of a system can be changed using the declarative and imperative approaches. For example, a software service is deployed and run on 2 containers on a physical node. Assuming that the service workload reaches a certain threshold, we therefore aim to deploy and run 4 more instances of the service; however, the capacity of the physical node is insufficient for this scale-out request. In a declarative infrastructure configuration system, we provide the orchestration system with our desired state, which is 6 containers of this service. The orchestrator, under the hood, picks a new physical node, sets up the networking components, instantiates the containers, and makes sure our system state ends up with 6 containers. However, in an imperative system, the client is supposed to instruct the orchestrator for each of these steps. The former approach abstracts away the complexities, but the latter approach enables the infrastructure to benefit from a degree of customisability.

Declarative approach. In a declarative system, the client is aware of a desired state, and the system is provided with a representation of this state, through which it can come up with a set of instructions to reach that state from the current state [194]. In Kubernetes, this approach is managed by various controllers. A representation of the pods is sent to the API server. After a few security checks, it stores the resource in the `etcd` database. The scheduler afterwards performs pod placement process based on this information. Based on scheduling decisions, the kubelets are contacted to start the containers. If more pods are planned to be started, that does not happen simultaneously.

Imperative approach. In an imperative system, the client is aware of a set of instructions to bring the system to a desired state [194]. Kubernetes operates as a declarative system through the API server. An imperative approach would bypass many steps such as scheduling and operate by communicating directly with the container runtime (e.g. the Docker daemon) on the nodes. That means pods can now be created in parallel if it is required.

2.2.3 Cold Start

When a serverless application serves an invocation request, or a microservice scales out due to a particular workload, one or more containers get created and started. To achieve this, the system requires to (i) bootstrap containers, (ii) prepare the software environments, and (iii) initialize the user code [33, 107]. This is called cold start. The execution of these steps might cause latency due to memory footprint, runtime, code package size, and more [38].

2.3 Related Work

We present the related work in two groups: (i) rapid deployments where the focus is on speeding up the container ecosystem, and (ii) queue-based approaches where different techniques based on pools of containers are researched.

2.3.1 Rapid deployments

Caching and snapshots. Cadden et al. [40] present a technique based on dependency planning by introducing two caching solutions to improve the startup time of a task. In brief, their cache-aware scheduler schedules tasks by creating containers on nodes where there is an exact container image cached, or a sub-layer of it. Although unikernels are inherently different in comparison

to Docker containers, Fu et al. [69] introduce an approach based on creating snapshots of unikernels ready to execute functions. Upon a function invocation, unikernels request the snapshots of this function to speed up its invocation.

Lazy image loading. In this approach, it is shown that only a small part of the container images (e.g. 6% in Docker images [90]) is enough to start a container, and the rest can be loaded lazily. FogDocker [54] presents a base image, after an analysis process on an image, with the essential files required to boot the container. The rest of the original image will be downloaded asynchronously. However, it is hard to mitigate application crashes at runtime when a required file is not yet fetched. Likewise, Slacker [90] uses the same technique mixed with layer-based snapshots and lazy cloning, i.e. this means that if Slacker wants to clone a particular layer of a Docker image, it only clones that specific layer and not the whole image. The advantage of this approach is that the pushing and pulling of containers run more smoothly, and as a result, the start-up time is reduced. A small trade-off is that the full runtime of applications becomes longer in experiments with a large load.

These techniques are not always compatible with each other. Some of the approaches based on Docker images are not always feasible in every situation, e.g. the *Lazy* approaches [54, 90], where we do not directly download all files from an image. This can become problematic with large applications that use a lot of files to operate.

2.3.2 Queue-based approaches

These techniques aim at preparing the infrastructural components (e.g. containers) as early as possible and keep them in pools. A warm queue of containers is a queue in which the containers are ready to process clients' requests. Lin et al. [122] reduce cold start latency by 85% through employing a pool of warm pods using Knative. However, the pod migration takes 2 seconds in this approach. Likewise, McGrath et al. [136] use cold queues to monitor the memory capacity of the worker nodes to start up new containers, and warm queues to keep track of existing warm containers. The other approach is to use pre-warmed queues where containers are started but the application is not yet initialized [192]. However, this technique is appropriate for stateless applications, especially those that do take much time to initialize, e.g. scripting languages like Python rather than compiled runtimes like Java and .NET [38].

The warm and pre-warmed queues are effective at the cost of high memory consumption. Mohan et al. [144] present an approach based on reusable network containers in OpenWhisk, inspired by `pause` containers in Kubernetes. In this way, they reduced the cold start latency by skipping the networking step.

Chapter 5.2.1 presents a complete overview of the cold start mitigation techniques at all layers of the infrastructure.

2.4 An imperative approach to cold start

In this section, we present three strategies, namely using (i) a pool of reusable network containers, (ii) a layered-based library sharing, and (iii) imperative configuration of application containers, to mitigate cold start problem in Kubernetes.

2.4.1 Reusable network containers

In this approach, we aim at pre-creating the network infrastructure of application containers to decrease the cold start time. To achieve this, a queue of warm network containers can be used. When scaling-out is required and a new application container must be started, our scaler binds the newly started container to a warm network container. Afterwards, the network container is removed from the queue and placed in a hot queue, meaning that it is occupied. Using the hot queue, the network container can be released and put back in the network container queue when the application container is no longer required. That ensures *reusability*.

In Kubernetes, network containers, also known as `pause` containers, cannot be created on a stand-alone basis. To have a queue of `pause` containers, our approach is based on creating lightweight pods. Each pod is composed of a `pause` container as a result.

Pod injection. When a scale-out request arrives, an application container is injected to a pod as follows: our scaler controller starts an application container; it picks an entry from the queue (i.e. the entry includes the `pause` container ID, `IPC` namespace and `cgroup` of each pod in the queue); it configures the container to use the network namespace, `cgroup` and `IPC` namespace of the pod; and it places the pod in the hot queue to know that the pod is already running a worker instance. We call this "pod injection" as an application container is injected into a pod without using the Kubernetes API, and instead by communicating directly with the worker nodes.

2.4.2 Layered-based Library Sharing

Layered-based library sharing is a technique with which external components and software dependencies of an application are encapsulated in various layers of container images to mitigate the cold start problem. Other work showed that library sharing reduces the memory footprint by sharing common portions of memory between containers [67]. Our hypothesis is that a layered-based approach has an effect on the startup time of an application. The previous research uses this technique to reduce the image size. That means if a layer with the library already exists, it can be used by multiple containers. This basically makes the container image (e.g. Docker) smaller for the second container, as it no longer needs to provide the libraries for its applications; therefore, it reduces the cold start.

Programming languages influence the startup time of applications [132]. For example, Java applications typically include most dependencies in JAR files. In a JAR file, there are application classes, libraries, frameworks and a manifest file. If we work with a large JAR file, we only have 1 layer containing the entire application. So it is impossible to benefit from library sharing. To speed up the container image retrieval, a JAR file can be split into multiple layers.

2.4.3 Imperative scaling in Kubernetes

Using an imperative approach in Kubernetes, our scaler communicates directly with the worker nodes, bypassing the Kubernetes controller components. In a declarative approach, a state change request (e.g. deployment of a pod) is processed by Kubernetes API, various controllers and schedulers to implement a control loop to mitigate unintended states (e.g. a node or container crash). Moreover, pods are started sequentially and placed one by one. But, our imperative approach communicates directly with the container runtime (e.g. the Docker daemon), start the application containers and inject them into the network pods. Therefore, the entire process can be done in parallel and at the same time. Our hypothesis is that it considerably improves the application start-up time when an application requires to scale out.

2.5 Evaluation

We aim to evaluate the effectiveness of the presented techniques, namely (i) library sharing, (ii) employing a pool of network pods, and (iii) the imperative approach, to mitigate the cold start problem. We also evaluate how our

imperative approach affects applications with strict SLO requirements based on job completion time.

Firstly, a job-oriented microservice application is introduced as our test application for the experiments. Secondly, we present our research methodology and the experiments conducted in that regard. Thereafter, we present our experiment setup and conclude with the experimental findings and their discussion.

2.5.1 Test application

We perform our experiments on a job processing microservice application (see Fig. 2.1). The application is composed of three parts: (i) the users who create jobs and add them to the queue, (ii) the queue where the active jobs are stored and (iii) the workers who are responsible for periodically checking the queue for jobs. Users add jobs to the queue. A job is a collection of tasks. When a lot of users are actively busy and producing different workloads, extra workers can be added to handle all possible requests. The `Queue` service and the workers have been implemented using RESTful Spring Boot in Java based on the Tomcat server.

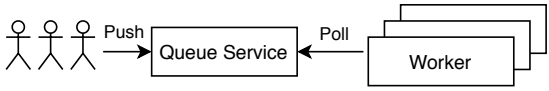


Figure 2.1: Test application

2.5.2 Experiment Methodology

Methodology. To evaluate the techniques introduced in this chapter, four deployment variations of the test application, presented in Section 2.5.1, are employed:

1. *DCL* where pods are created in a declarative way via Kubernetes and its policy-based scheduler, i.e. the pause and application containers are created together.

2. *IMP* where network pods are created in advance so that we can use them to inject our application containers, i.e. the injection of these application containers is done asynchronously by running a script on the node on which pods are available.

3. $DCL + LIB$ where everything is identical to DCL except now the Docker image of the application container consists of multiple layers to allow library sharing.
4. $IMP + LIB$ where everything is identical to IMP but with library sharing as $DCL + LIB$.

We employ the Locust [126] load testing tool to allow multiple users to join the experiments. They register jobs in the queue to start a scaling action. Each job that is being submitted by a user runs in its own container. We repeat these workload tests for each of the abovementioned deployment variations of the test application. Throughout this process, we inspect the occurrence of different events to measure the duration of: (i) starting the pause container, (ii) application container startup delay, (iii) starting up the application container, (iv) starting up the JVM, (v) starting up Tomcat server, (vi) finalizing Spring Boot startup, and (vii) finishing the first task. We analyze the effect of the deployment variations on these durations to understand how the different techniques of our approach affect the cold start problem.

Library sharing validation. Instead of using a large JAR file, we extract 3 layers to reuse in our Docker file. To speed up the build time of the Docker image, we have placed the layers that change the least, such as libraries, above the classes. If the dependencies are not changed, then the library layer does not need to change, resulting in a faster build time. The library layer can also be easily shared if we want to work with other applications.

No dependency layers

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD target/worker.jar app.jar
ENTRYPOINT exec java -jar /app.jar
```

Layered library sharing

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG LIB=target/
COPY $LIB/BOOT-INF/lib /app/lib
COPY $LIB/META-INF /app/META-INF
COPY $LIB/BOOT-INF/classes /app
ADD README.md README.md
RUN apk update java=8u191+
ENTRYPOINT exec java -cp "app:app/lib/*"
"be.kuleuven.WorkerApplication"
```

To validate that library sharing is properly implemented, we use the `pmap` and `smaps` tools as used in [67]. It allows us to check which libraries are being used, how much memory they share with the other processes, and obtain more detailed information about the memory usage. We create two containers with the same image. We perform this validation once with library sharing and once without. Our results show that the libraries such as Tomcat, Spring, Avalon, etc. have been successfully shared in the case with library sharing; however,

`pmap` and `smaps` shows no libraries in the *shared memory* section in the other case. We can therefore conclude that we have successfully shared our libraries with each other in our library sharing Docker container.

Experiment setup. To ensure that the results are comparable, we use the same environment for all experiments. This environment is an OpenStack-based private cloud. We created four virtual machines (VMs) where three of them belong to the Kubernetes cluster while the fourth one is used for load testing. The operating system of the VMs is Ubuntu 16.04. We used Kubernetes v1.12. The resource allocation of the VMs is organized as follows: (i) one Kubernetes master node with 2vCPU and 4GB RAM as the master node of the cluster, (ii) one worker node with 4vCPU and 8GB RAM. This worker node hosts both the test application and the scaler that is responsible for deploying the worker pods, (iii) one worker node monitoring with 2vCPU and 4GB RAM, and (iv) the load generator node with 4vCPU and 8GB RAM to simulate the load of an external user submitting jobs.

2.5.3 Experiments

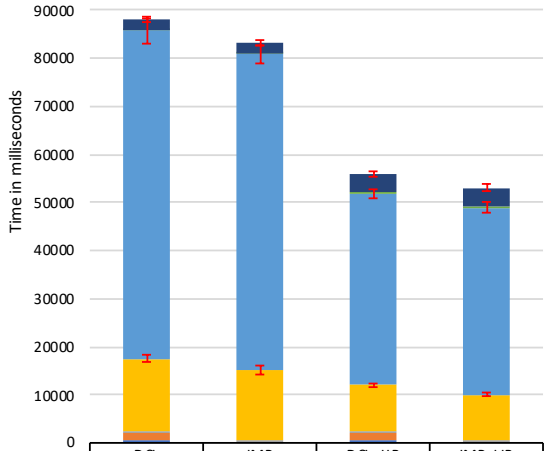
In this subsection, we present our experiments and our findings. We have performed two experiments:

1. *Experiment 1* to measure the total runtime of containers individually (i.e. 1 – 6 containers were created) and understand the impact of the presented techniques on cold start
2. *Experiment 2* to measure the start and end time of 6 containers with 1000 tasks, especially to understand the impact on a scenario where there is a strict Service Level Objective (SLO) such as *job return time*

Experiment 1

This experiment includes 6 rounds, and each round has been executed against each of the deployment variations, namely declarative (DCL), imperative (IMP), declarative with library sharing ($DCL + LIB$) and imperative with library sharing ($IMP + LIB$). In the first round, only one container is launched for one user, in the second round, 2 containers are launched for 2 users, and so on up to 6 containers/users. We inspect the start-up time of the pods with a granularity described in Section 2.5.2. We ran each experiment at least 20 times and then averaged these results. Only 10 experiments have been performed with the six containers. We also show the standard deviation error bars on

the graphs in red showing how much the values differ from each other. In our research, we investigate the extreme cases: Round 1 with one container, and Round 6 with six containers.

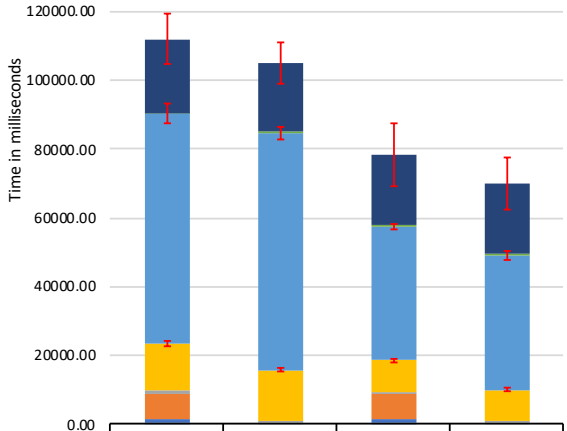


	DCL	IMP	DCL+LIB	IMP+LIB
Duration finish first task	2044.70	2130.70	3778.81	3867.10
Duration finalizing spring boot startup	154.10	169.90	396.57	304.80
Duration starting up Tomcat server	68230.95	65627.05	39771.05	38769.10
Duration starting up the JVM	15156.50	14801.55	9643.38	9639.75
Duration starting up the application container	434.55	463.60	436.10	480.85
Application container startup delay	1413.20	0.00	1426.57	0.00
Duration starting the pause container	606.80	0.00	592.71	0.00

Figure 2.2: The fined-grained start-up times and execution time of (round 1, container 1) when only 1 container is scheduled to be created

Round 1 (one container). Fig. 2.2 illustrates the first round in which only 1 container has been started (i.e. the results of registering one user). We collected fine-grained information regarding the startup of this container. We note that the strategies that use library sharing have a major impact on the boot time of the Tomcat server as well as when starting a JVM instance. The reason for this is that the libraries that Tomcat uses are provided with a higher layer in the Docker image and we therefore benefit from the layer-based library sharing that we introduced in Section 2.4.2. A small trade-off is the execution time of the first task with the library sharing strategies. We notice an increase when the Spring boot is completed.

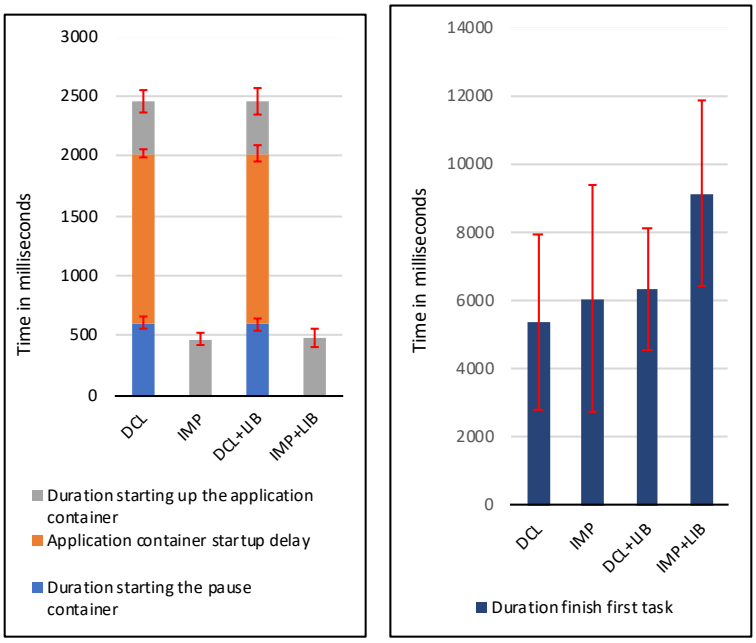
The imperative approach, in Fig. 2.2, does not seem to have considerable impact



	DCL	IMP	DCL+LIB	IMP+LIB
Duration finish first task	21498.18	19829.33	20608.80	20441.60
Duration finalizing spring boot startup	171.45	182.89	331.60	371.80
Duration starting up Tomcat server	66814.09	68945.44	38899.30	39053.90
Duration starting up the JVM	13897.64	14867.89	9097.70	9168.50
Duration starting up the application container	611.36	1041.78	618.60	997.70
Application container startup delay	7660.00	0.00	7503.40	0.00
Duration starting the pause container	1401.82	0.00	1349.50	0.00

Figure 2.3: The fined-grained boot-up and execution time of (round 6, container 6) when 6 containers are scheduled to be created

on the whole picture. That is because the Java ecosystem, and most compiled runtime enterprise languages, are slower at startup in comparison to scripting languages such as Python and Ruby. However, to figure out the real advantage of creating the network containers in advance, we zoom in on the bottom events of this chart regardless of the application deployed within the container. Fig. 2.4a illustrates a closer look at the container-related startup events. In this figure, the impact of creating a network container in advance is more clearly indicated. We can understand, that thanks to the creation of a network container in advance, we gain about two seconds speedup. This is certainly an improvement, as it does not depend on the programming language and the application dependencies. We obtain these results since with the imperative approach the `pause` containers were picked up from the pool of pods. Moreover, the declarative approach typically create the `pause` containers first and then the application containers; therefore, there is a delay in between, which is avoided in the imperative approaches as a consequence. In Fig. 2.4b, we zoom in on



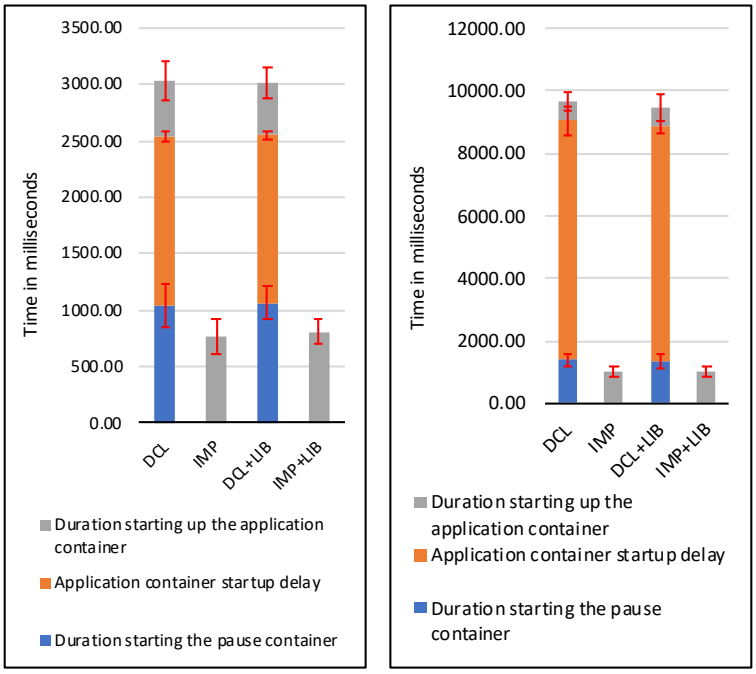
(a) The container startup events, (round 1, container 1) based on Fig. 2.2

(b) The duration of completion of the first task (round 2, container 1)

Figure 2.4: A closer look at Experiment 1 (round 1 and round 2)

the duration of the first task and notice that the standard deviation error bars overlap. When the standard deviation error bars do not or almost not overlap, the results are significantly different. In our case, however, we observed that the durations of the first task completion were not significantly different in this experiment.

Round 6 (six containers). Fig. 2.3 shows the timings of the different deployment variations for the sixth round where 6 containers are started for 6 users. We performed the same experiments as in to Round 1, but we seek to see the differences or correlations with the previous observations in the other rounds. In this figure, we observe that the library sharing technique continues to have a major impact on the boot time of the Tomcat server and JVM. The duration of starting up an instance of the Tomcat server is even almost halved. However, the time of "finalizing spring boot startup" is doubled, but the numbers are only in the range of ~150 milliseconds, which is little compared to the benefit we get from the speedup of the other dependencies (Tomcat and JVM).



(a) The container startup events, (round 6, container 1)

(b) The container startup events, (round 6, container 6)

Figure 2.5: A closer look at Experiment 1 (round 6)

In Fig. 2.5a, the startup times from Fig. 2.3 are illustrated separately in which 6 containers (pods) have been started. Here we can deduce that creating network containers in advance has a minimal impact. However, if we do this together with an imperative configuration, we notice a great improvement. The advantage is greatest with successive containers. We notice this especially with container 6 of this experiments. The reason is that Kubernetes needs to wait for all `pause` containers and the predecessors to be created; that introduces considerable delay. More precisely, to create 6 pods, Kubernetes starts with creating the pause containers one by one. Once all the pause containers have been created, it starts with creating the application containers one by one. For instance, if we want to know the total startup time of the 4th pod with the *DCL* approach, application container 4 needs to wait till all pause containers (1...6), and the application container 1, 2 and 3 are created and started. Then, the container 4 can get created and started; by starting up we mean the container not the application. The only drawback of the imperative approach is that the duration of "starting up the application container" increases. But it is a small trade-off.

We can conclude that the imperative approach is considerably beneficial when multiple containers get started up simultaneously.

Findings. Based on the results of the experiment 1 (see Section 2.5.3), several findings can be concluded as follows:

1. The library sharing technique has the greatest impact across all experiments. It resulted in a reduction in the start-up time of the JVM and the Tomcat server. The only drawback is that the duration of "finalizing spring boot" is getting longer, but this is very little compared to the overall improvement.

2. The library sharing technique has a negative impact on the execution time of the first task. However, the standard error bars overlap, which indicates that statistically the results of these experiments are not significantly different. This means that library sharing across multiple experiments does not negatively impact the execution time of the first task.

3. The approach to pre-create the network containers seems to have a small impact on cold start when starting only one container. The only advantage is that we can skip the network creation step. However, the impact is greater when multiple containers start up simultaneously. The reason for this is the way Kubernetes creates its pods and containers. Firstly, it creates the necessary pause containers before the application containers. As a result, we observe a considerable impact on the boot-up time, especially with the last container. In the Kubernetes approach, the last container has to wait until all pause containers and application containers have been created before it can start its own creation process. A small drawback with this technique is when several containers are started simultaneously, the start-up of the application container takes longer. However, this is again negligible in comparison to the overall improvement.

Experiment 2

In this experiment, we primarily focus on the start time of the application containers; creating the network containers in advance does not affect this experiment. The aims of this experiment are (i) to examine the possible impact of different strategies on the start and end time of the application, (ii) to realize which technique is more suitable in case of having service layer objectives (SLO) such as job completion time, and (iii) to understand the impact of each approach on the CPU usage. We expect that the strategies that start their containers the

fastest will also reach their optimal CPU usage faster. We test the techniques on the aforementioned deployment variations of the test application discussed earlier.

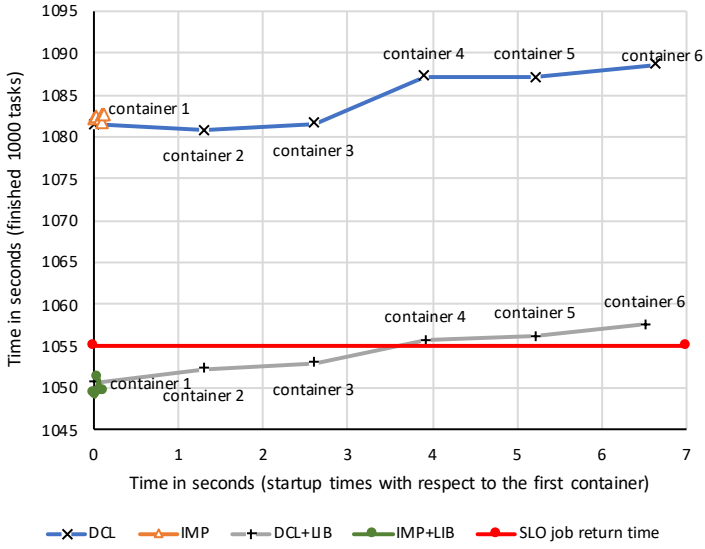
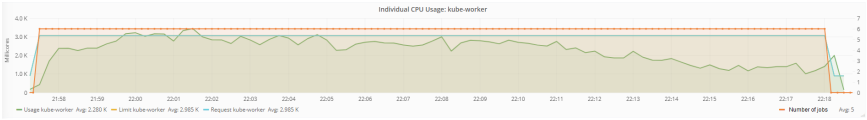


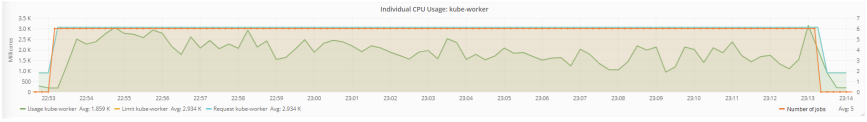
Figure 2.6: Experiment 2 with 6 containers

We let six users register in the application simultaneously. After that, each user sends 1000 tasks to the application. The end time is equal to the time when a user finished his 1000 tasks. We use Locust to simulate the workload generated by the users. To determine when the 1000th task was executed, we look at the logs of the Docker application to extract the time of the 1000th task. The start and end time of each case are measured in relation to the start time of the first container. In other words, if container 2 has started three seconds after container 1, the startup time of container 2 is three seconds.

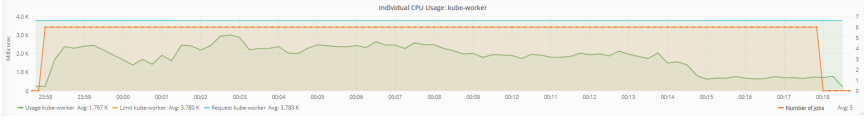
Impact on the duration of processing all tasks. Fig. 2.6 illustrates that the imperative approach (IMP and $IMP + LIB$) starts all 6 containers before the start of container 2 of the ones with declarative approaches (DCL and $DCL + LIB$). In each experiment, the completion times are illustrated as overlapping dots on each other for all containers. This is due to the asynchronous creation of the application containers using the imperative approach. Therefore, it means that the other containers do not have to wait for container 1 to initiate their own start-up process. Moreover, our library sharing approach has a positive impact on the end times of this experiment. This impact comes from



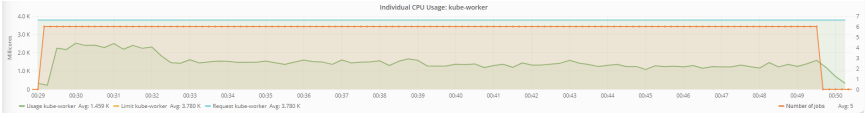
(a) The declarative approach without library sharing



(b) The declarative approach with library sharing



(c) The imperative approach without library sharing



(d) The imperative approach with library sharing

Figure 2.7: CPU usage in Experiment 2

the previously seen benefits of faster boot times for the Tomcat server and JVM (see Experiment 1).

Meeting job completion time objective (SLO). In Fig. 2.6, the red line represents the job return time as an SLO. An SLO is an agreement that is made between the service provider and the customer. The agreement is that 1000 tasks will be performed per container in 1055 seconds. If we look at our imperative approach, we conclude that this deadline is easily achievable by all containers. With the declarative approach, this SLO is no longer feasible for the fourth container. This is because the 4th container has to wait for the creation of container 1, 2, 3 and all pause containers. We can deduce that our imperative approach is effective to meet SLO deadlines because these containers can start up in parallel.

Fastest containers reach optimal CPU usage faster. The cold start problem also affects CPU usage. This problem causes the CPU usage to increase steadily as shown in Fig. 2.7a. If we look at the impact of library sharing in Fig. 2.7b, we can already observe an improvement. We see a straight rise to the highest point. Fig. 2.7c shows the effect of the imperative approach. The start-up phase of this experiment shows an improvement in the CPU usage,

but it eventually fades. This is because this approach is mainly effective for the start-up of the containers. Finally, we combine these two strategies in Fig. 2.7d, where the two previously seen benefits are combined. Our CPU usage is now rising in the beginning and right to its peak.

Findings. Based on the results of the experiment 2, our findings are as follows:

1. The imperative approach further enables the application to meet SLO targets without a non-deterministic delay that appears in the declarative approach when multiple Pods must be created at the same time due to concurrent user requests.
2. These two techniques not only reduce the cold start but also provide the application with the processing power (CPU) as early as possible, resulting in a faster overall job completion time.

2.6 Conclusion

We investigated how a queue of reusable network containers, layer-based library sharing, and imperative configuration management, when combined together, can improve the *cold start* problem in Kubernetes. Cold start is a well-known problem in the elastic scaling of containers, especially in serverless computing. Sometimes several containers are required to be started simultaneously, and the applications deployed on these containers are the same or share software dependencies, increasing the impact of this problem. We evaluated the above-mentioned techniques extensively in a deadline-oriented job processing microservice.

Our findings show that (i) the library sharing approach results in a large reduction in the start-up time of software dependencies (e.g. the JVM and Tomcat server), (ii) pre-creating network containers has greater impact when multiple application containers are started in parallel, (iii) the imperative configuration approach introduces start-up time determinism and predictability, making this approach more reliable for applications with SLOs such as job completion deadlines.

Chapter 3

InfraComposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation and optimization workflows

In this chapter, we present a policy-driven adaptive and reflective middleware that supports smart cloud-based deployment and execution of engineering workflows. This middleware supports deep inspection of the workflow task structure and execution, as well as of the very specific mathematical tools, their executions, and used parameters. The reflective capabilities are based on multiple meta-models to reflect workflow structure, deployment, execution, and resources. Adaptive deployment is driven by both human input as meta-data annotations as well as adaptation policies that reason over the actual execution history of the workflows. We validate and evaluate this middleware in real-life application cases and scenarios in the domain of aeronautics.

This research is based on a publication in the Journal of Systems Architecture [16] in 2019. Prior to the journal paper, a subset of this work has been published at the 16th Workshop on Adaptive and Reflective Middleware (ARM'17) [15] as well as a publication at the European Conference of Simulation Process and Data Management (NAFESM'18) [133]. The rest of this chapter is

structured as follows. Section 3.1 introduces the overall context and challenges of deploying and executing engineering workflows on the cloud. Furthermore, we present a high-level description of the InfraComposer middleware and its contributions. Section 3.2 presents two motivating scenarios of engineering workflows and their common patterns. Section 3.4 describes the architecture and the concepts of the middleware. Moreover, we introduce the meta-models, policy-driven architecture, and monitoring components. Section 3.5 validates multiple adaptive (re)deployment scenarios and demonstrates the effectiveness of the InfraComposer policy-driven architecture. Section 3.3 presents the state of the art in cloudification of workflows, adaptive middleware, and auto-scaling techniques. Section 3.6 outlines the limitations and the possible opportunities to extend the current work to become smarter and more comprehensive. Section 3.7 concludes this chapter and outlines our research outcomes.

3.1 Introduction

Engineers in major industries, such as aerospace and automotive, use simulation and optimization workflows to create, simulate and optimize complex designs. Such workflows are complex and long-running processes, which are typically composed of various software tools and services, to simulate and optimize physical properties such as strength, vibrations, geometrical decomposition or material selection. These tools can be knowledge engineering tools such as ParaPy, and engineering tools such as MathWorks, Cradle, and more. Engineers use different hardware to execute these workflows, e.g., their desktop computers or High-Performance Computing (HPC) clusters.

Current situation. Desktop computers have limited capacity in terms of processors, memory, and storage. In addition, the parallel execution of the experiments is tied to the number of available cores and computers. HPC clusters, unlike desktop computers, are very efficient and powerful, but they are constructed with dedicated and expensive hardware, and their capacity is not always directly available. Moreover, time slot reservation and complex queuing API are yet another hassle for those engineers with long-running or recurring experiments.

The promise of the cloud. Engineers can nowadays benefit from cloud computing to gain on-demand access to the required resources for their workflows, often based on cheap commodity hardware. Cloud computing is a model for enabling *on-demand* network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) [140]. Cloud orchestration tools enable automated provisioning of

the required cloud-related resources such as virtual machines, virtual networks, and required infrastructure software and middleware platforms. Therefore, the infrastructure and deployment processes have become completely *composable and programmable.*

Challenges. There are still key problems and challenges when deploying and executing engineering workflows in the cloud. Engineers need to automate the deployment, as well as support smart scaling and execution of simulation and optimization workflows in the cloud. For each deployment and execution of the workflow, this process includes adaptive deployment to collocate, separate and parallelize the different tasks and their specific tools on the right amount and the right type of nodes. Both can also vary depending on the specific parameters for a certain execution. Automating this process includes automatic determination of the required resource types (virtual machines, storage volumes, etc), automatic estimation of the amount of cloud resources (number of virtual machines, amount of memory and cores, etc), as well as the automatic bootstrapping and destruction of the required infrastructure.

InfraComposer: towards smart deployment in the cloud. To address these challenges, we present a **reflective and adaptive middleware** that enables and manages (i) smart, adaptive workflow deployment, (ii) scaling and (iii) execution in the cloud. We leverage both the domain-specific knowledge about the concrete tools that are used, and deep inspection of these tools when deployed and executing on the cloud platform.

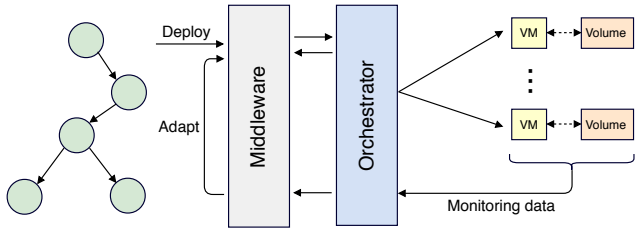


Figure 3.1: Middleware for cloudifying simulation workflows

The **adaptive middleware** is driven by both inputs from the engineers about the properties of the tools they use, as well as the execution history of these tools. The input from the engineers is specified as annotations on the workflows and is based on human knowledge and assumption about the tools with regards to CPU usage, memory usage, and network usage. The execution history over time will be used to optimize the original deployment and scaling plan, and thus to further adapt to actual real execution knowledge (see Fig. 3.1).

As such, our middleware defines two key contributions to existing orchestration and deployment middleware:

1. *Annotation-driven resource reservation and deployment planning.* Based on the annotations in the workflow, an initial deployment plan will be generated. A deployment plan is a topology and orchestration specification for a cloud application which can be executed by a cloud orchestrator.
2. *History-driven adaptive scaling and reconfiguration.* The middleware adapts the configurations of the deployment plans based on the execution history of the workflows using previous results. This is driven by policies that can reason about, and perform statistical analysis on the execution history.

To achieve this, the **reflective meta-models in the middleware** enable reification of:

1. key architectural concepts such as workflows, tasks, specific tools and their deployment,
2. key execution concepts such as specific tool executions with specific parameters, and
3. key resource utilization concepts such as nodes, cores, CPU time, memory, storage, and network metrics.

Moreover, we validate and evaluate the concepts with real-world, industrial use cases and scenarios reflecting actual production settings (see Section 3.2). The main focus of this work is to present a holistic cloudification system for engineering workflows with special attention to the reification of various concepts. We validate the applicability of the system and relevance by implementing and analysing these engineering use cases in general, and in particular aeronautics.

First, we extend the middleware concepts with policy-based adaptation. Second, we describe the fine-grained architecture of the adaptive and reflective middleware. Third, we provide extensive validation and evaluation in multiple real-life application cases and adaptive scenarios.

3.2 Motivation and Use Cases

In this section, we present two examples of the industrial workflows from the aeronautics domain. The goal is to discuss the adaptive scenarios.

Electrical Wiring Interconnection System (EWIS) In the design process of aircrafts, EWIS is an important step of multidisciplinary design optimizations (MDO). Aerospace engineers design and execute the complex simulation and optimization experiments to find optimal solutions for cockpit wire harness routings.

Motivating Scenario. When considering the tasks in an EWIS workflow, it is unclear whether the main optimization tool used by a task is memory intensive. The most optimal deployment is achieved in a step-wise and adaptive process: first driven by the engineer's annotations, and then by the execution history of the tool.

(1) The aerospace engineer specifies that the wire harness tool is not CPU intensive but memory-intensive, because it loads a very large amount of data, e.g. physical features of an aircraft, in memory. At least, this is the engineer's assumption, defined as annotations. (2) The middleware allocates a large amount of memory with few cores for each virtual node in the cloud and enables the engineer to execute the experiment as specified in the workflow. (3) However, execution history shows that the tool is mostly CPU intensive because the executing nodes reach the system load saturation limits, and the memory resource has been overallocated. (4) The deployment plan should be updated to employ a higher number of cores and less amount of memory for the virtual nodes.

Design of the hinge system of an aircraft rudder Another multidisciplinary design optimization (MDO) example can be found in the design process of aircrafts. Optimization of a hinge design is a crucial part of an aircraft rudder design as a whole, and it is automated as a workflow. The workflow consists of a set of engineering tools, which are responsible for meshing and stress analysis of hinge components, as well as performing a quasi-exhaustive search for all different possibilities in order to minimise engineering objectives (e.g., total weight) [97, 114]. In Section 3.5.2, we present a workflow as a validation.

Motivating scenario. These engineering tools are interdependent, and their execution flow within the workflow is based on complex designs developed by MDO experts. Therefore, the supported level of parallelization, as well as the number of nodes and the required tool instances are unclear upfront to an aerospace engineer.

(1) First, the engineer of the workflow, via annotations, specifies that only a very large node is required. Moreover, the engineer also specifies an expected execution completion time. (2) The middleware instantiates the node and executes the experiment as specified in the workflow. (3) However, the execution history shows that the execution takes much longer than the initial anticipation due to a large number of sub-experiments. This results in many parallel runs and scheduled jobs. (4) The deployment plan should be updated to adjust the number of nodes (rescaling) with regard to the expected execution time.

Section 3.5 presents more scenarios. Based on each of these workflows, a similar pattern emerges:

- The deployment middleware needs initial domain knowledge about the tools to achieve initial deployment plans.
- Taking the input parameters of the tools into account, deployment plans should be optimised to achieve optimal executions.
- These workflows are typically composed of various discipline analysis tools for execution. Each tool can be installed on different operating systems and on specific host types (memory-focused host, CPU-focused host, or high-performing storage hosts with SSDs).
- These workflows are often computationally intensive, and their execution sometimes takes hours, days or weeks to be completed. Engineers use parallel runs to speed up the execution.
- These workflows go through continuous improvement by reconfiguration of the design parameters. Engineers re-execute the improved versions recurringly to optimize their objectives. Therefore, the varying input parameters of the workflows might influence the system in a way that the deployment plans require adaptation accordingly. Because tools, for example, might become more dependent on CPU than disk for different parameters.

These common patterns introduce several key problems and challenges (refer to Section 3.1) leading to the manual, duplicate, complex, and time-consuming work for the engineers.

3.3 Related Work

Execution environment reproducibility Reproducibility and repeatability of workflow execution environment are crucial aspects in scientific and engineering workflow deployment. In that regard, Santana-Perez et al. [174] describe the execution environment of workflows using semantic vocabularies to produce annotated workflows (i.e., logical preservation of execution environment). TOSCA [195] is an OASIS standard to describe the topology of cloud-based applications towards portable, reproducible application deployments. Qasha et al. [167] combine two execution-environment reproducibility techniques (i.e., the logical and physical preservation) of scientific workflows using TOSCA in a container-based approach. In addition to the plain reproducibility concerns, our middleware architecture employs reflection concepts to reconfigure deployment plans, resulting in *efficient* execution environments.

Reflective middleware The concept of reflection was first introduced by Smith [183] in programming languages. Other works [23, 171] adopted the concept in the middleware platforms, focusing on reconfigurability and openness of such systems. Blair et al. [23] present two types of reflection: *structural reflection* and *behavioural reflection*. Structural reflection is concerned with the structure and the content of the component, which is represented by two distinct meta-models, namely the encapsulation and composition meta-models. Behavioural reflection is concerned with activity in the system, which is represented by the environmental meta-model. Existing literature [22] extends the architecture by presenting the resource meta-model to address the reification of resource management. Weyns et al. [202] present a comprehensive reference model for distributed self-adaptive systems with a special attention to the reflection perspective. As applications of the reflection concepts in web services, [72, 14] present adaptive and reflective middleware systems which are able to expose their functionalities to application developers. Our meta-models are inspired by these studies.

Auto-scaling of cloud resources Deployment plans are reconfigured by rescaling of resources based on execution history, either horizontally or vertically. A recent survey by Chen et al. [49] classifies the decision making tactics of the self-adaptive cloud autoscaling systems into three major approaches: (i) rule-based control, (ii) control theoretic approaches and (iii) search-based optimisation. Alternatively, Lorido et al. [127] categorise auto-scaling techniques generally into *reactive* (i.e., based on rules and current data) and *proactive* (i.e., based on prediction) approaches, as well as a more fine-grained classification, resulting in: (1) threshold based rules, (2) reinforcement learning, (3) queuing theory, (4) control theory, and (5) time series analysis. Recently, Dhuraibi et al. [6] conducted a comprehensive survey on elasticity in cloud computing with a special attention to containers. InfraComposer employed a policy-based, history-driven approach akin to the threshold-based approach. Thresholds are statically defined, similar to the other existing literature [135, 62, 87, 86]. Most related work [62, 87] use single or multiple metrics, but Hasan et al. [91] employed several metrics from several domains such as compute, storage and network. InfraComposer also forms its policies based on multiple domains varying from workflow primitives to various cloud primitives.

Scientific workflows in the cloud Among open challenges of migration and execution of scientific workflows on the cloud [208], computation and data management are crucial. Processing large scientific data has impact on execution mechanism of the workflow engines. Kacsuk et al. [108] present efficient data pipelines by using a service choreography concept instead of the enactor-based

workflow concept. Furthermore, data locality has influence on performance and overall execution time [188, 41]. Regarding adaptive execution of the workflow in the cloud, Oliveira et al. [155] introduce an adaptive approach to dynamically tuning the workflow activity size to achieve better performance, and Wang et al. [196] present an adaptive workflow management through dynamic iterative optimisation framework. Although engineering workflows are inherently different in comparison to scientific workflows, some of the challenges regarding data management and adaptive execution share common requirements and concerns.

3.4 The InfraComposer Middleware

This section presents the architecture of InfraComposer, focussing on the different features and subsystems of the policy-driven reflective and adaptive middleware for the cloudification of engineering workflows.

First, the middleware supports annotation-driven, cloud-based deployment of engineering workflows and their different subtasks. During execution, it collects runtime information about task executions and the underpinning infrastructure by reflective monitoring of resources, as well as deep inspection of software tools. Adaptation policies, which are based on the execution history, enable the middleware to reconfigure the deployment plans to be adaptive for recurrent execution of the workflows .

The InfraComposer middleware architecture consists of four main components (see Fig. 3.2): (i) a *workflow manager* component to expose a workflow deployment API and to identify the annotated tasks and their annotations, (ii) a *configurator* component to generate configurations based on given annotations with respect to execution history data, (iii) a *deployment plan composer* component to produce deployment plans based on elementary deployment modules for the cloud orchestrator and to initiate the deployment, and (iv) a *monitoring* component to store live monitoring data of workflow execution.

The rest of this section is structured as follows. First, we describe the annotation-based deployment. We then elaborate on the reflective capabilities and the meta-models. Third, we describe the policy-driven adaptation architecture.

3.4.1 Annotation-based Deployment

A simulation and optimization workflow is a group of tasks that, once completed, will accomplish some objectives. As explained in Section 3.1, these tasks employ different analysis tools, which are responsible for the execution. Workflows and

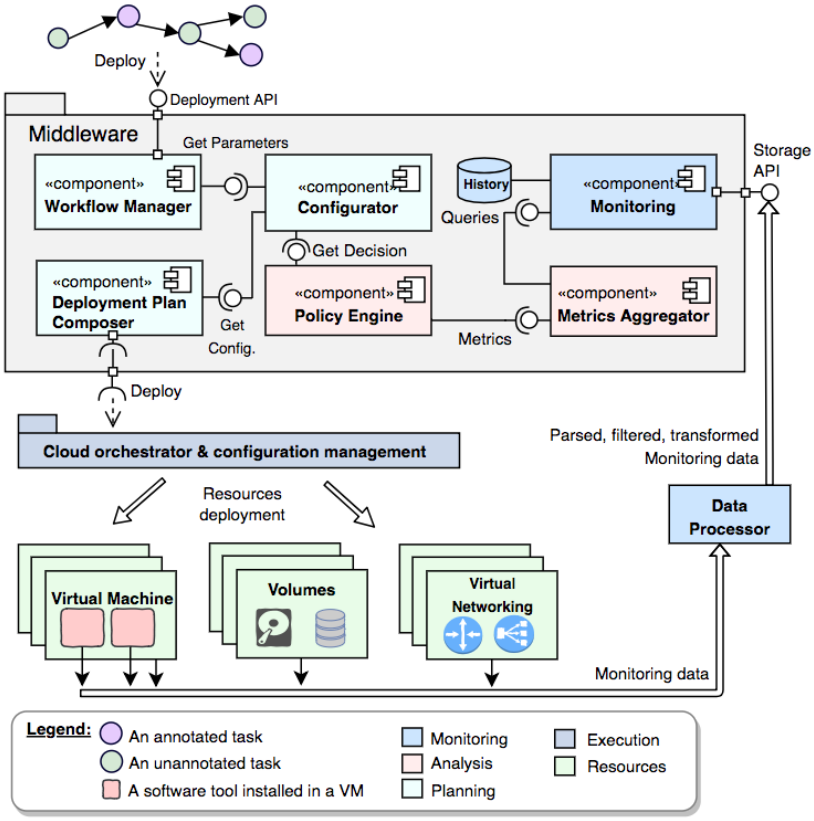


Figure 3.2: Overview of the middleware with the annotation processing and configuration components.

tasks can be annotated to provide more information about the required resources. InfraComposer is capable of identifying these annotations in the *workflow manager* component to provide necessary information for the *configurator* component. There are two categories of annotations: direct-deployment and resource-consumption annotations.

Direct-deployment annotations. These annotations provide the middleware with direct information concerning deployment of workflows on the cloud. The crucial aspects described by annotations are the employed analysis tools, their deployment locations and number of instances. Tools can either be collocated or separately deployed on the nodes. For example, annotations could describe that some tools should be deployed on one node, others on individual nodes, and there should be five instances of each node. Furthermore, network-related annotations can propose a networking scheme for tools and nodes where necessary. For

example, network-level segregation of nodes can be achieved for a network intensive workflow.

Another direct-deployment annotation is the identifier of the existing resources (e.g., instance images, volumes, networking components, etc.). For instance, some of the engineering tools need human involvement during the installation process, or install slowly due to the size of the packages. Therefore, these tools can be installed and prepared as virtual machine images to speed up the deployment process. The unique identifiers of the images help the *configurator* component provide configurations to the *deployment plan composer* component.

Resource-consumption annotations. These annotations provide general, approximate information about the infrastructural resource requirements of the tools with regard to disk, memory, processor and network. The four main categories of resource-consumption annotations are presented in Table 3.1.

Category	**Annotation**	
Disk	Disk intensive	percentage
	Required disk space	GB
	Data locality	location
Memory	Memory intensive	percentage
	Required RAM	GiB
Processor	CPU intensive	percentage
	GPU intensive	percentage
	Required cores	number of cores
Network	Network intensive	percentage
	Required bandwidth	Mbps

Table 3.1: Four main categories of resource-consumption annotations.

The computationally intensive workflows should employ suitable virtual machines in order to execute efficiently and to minimise the interference with other co-existing cloud users. Some virtual machines share the physical processors with other tenants, and some have CPU-pinning, meaning that the virtual cores are mapped to the physical cores in a shared-nothing approach. Moreover, some public cloud providers offer domain-specific types of virtual machines such as accelerated instances with GPU.

Furthermore, parallel execution of workflows may have considerable network overhead due to the continuous transfer of large chunks of data. That can easily saturate the bandwidth, slow down the execution, and interfere with other co-existing users. Network annotations enable the middleware to compose appropriate network architectures based on the available networking infrastructure.

Regarding other potential annotations which are not evaluated in this thesis, workflows can specify whether the experiment is disk intensive, as well as the required space. Cloud providers, either private or public, offer various types of storage systems with varying capabilities, speed, etc. In addition, some clients are concerned about data locality due to the enterprise policies or governmental law (e.g., GDPR[65]). Such annotations can potentially guide InfraComposer to select appropriate storage options with respect to the given constraints.

3.4.2 Reflective Capabilities and Meta-models

There are four styles of reflection in InfraComposer represented by four meta-models, namely structural, deployment, execution and resource reflection.

Structural reflection. Structural reflection [81] results in a meta-model of the different static concepts in the workflows defined by the engineers, which represents the structure of workflows and tools within the middleware and the execution history. Fig. 3.3 illustrates the meta-model. This meta-model describes engineering workflows and composition of activities and tools, as well as annotations.

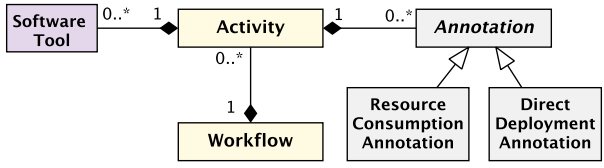


Figure 3.3: The structural meta-model.

Deployment reflection. Deployment reflection results in a meta-model representing and reifying the concepts in the deployment model. Fig. 3.4 illustrates the deployment plan and the mappings between workflows, activities, tools (not illustrated), and cloud-based components such as compute nodes, storage and networking elements.

Execution reflection. Execution reflection results in a meta-model of the execution of activities and specific tools on specific nodes. This model reifies concepts such as execution per workflow, execution per activity, and execution per tool with respect to the engineer's given design parameters (see Fig. 3.5).

Resource reflection. Resource reflection results in a meta-model of the underpinning cloud infrastructure resources and domain resources (see Fig. 3.6). Cloud infrastructure resources include concepts such as processing (cores), compute nodes, memory, network and storage, which are reified for each execution by consumption pattern. Such reflection allows the monitoring

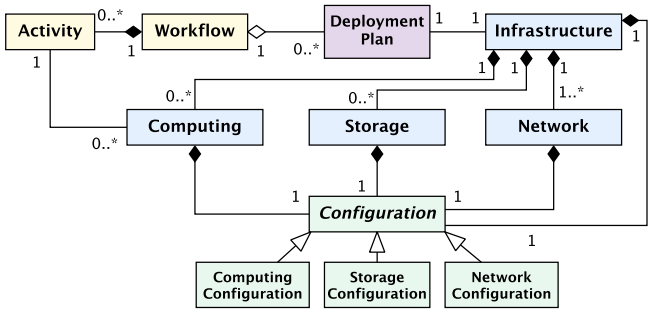


Figure 3.4: The deployment meta-model.

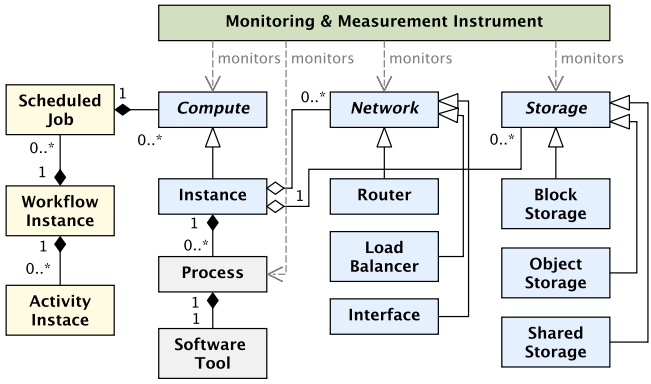


Figure 3.5: The execution meta-model.

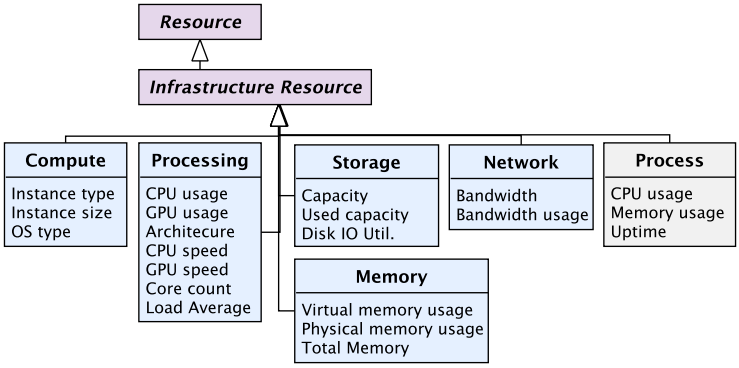


Figure 3.6: The resource meta-model.

and adaptation phase to benefit from coarse-grained or deep introspection of resources, leading to more efficient resource allocation in the future execution of the workflows.

3.4.3 Policy-driven Adaptive Architecture

InfraComposer monitors the execution of the workflows and builds a database based on the execution history, with which it fine-tunes the deployment plans and the configurations to make the future executions more efficient. Adding intelligence to the smart adaptation capabilities of the middleware requires the acquisition of new knowledge about the execution of the different tasks and tools. For example, an annotation suggested that a tool was disk intensive, but the execution history indicates that it is CPU intensive and only uses two cores for input files smaller than 100 MB.

A recent survey [49] classifies the architectural patterns of self-adaptive autoscaling systems into three groups: feedback loop, observe-decide-act and MAPE(-K). InfraComposer is a policy-driven (self-)adaptive middleware following the MAPE-K [110] architecture of self-adaptive systems [129] (see Fig. 3.7).

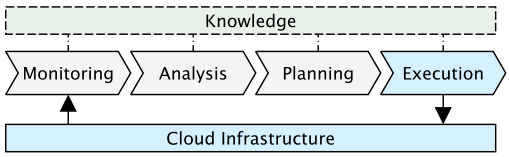


Figure 3.7: Adaptive middleware using MAPE-K control loop.

Among different architectural patterns and decision making approaches, we employed MAPE-K on account of better separation of concerns. Furthermore, we opted for a *rule-based control loop* because of negligible overhead and manageable policy reconfigurations at runtime.

As illustrated in Fig. 3.2, the *monitoring* phase collects information from the cloud and workflow-specific resources. The data processor component parses, filters, and transforms the monitoring data and persists it in a time series database (TSDB). The *analysis* phase uses the persisted monitoring data by performing statistical aggregate queries (e.g., max, min, average, percentile, etc.), and it employs a policy engine to enforce adaptation policies to assess the previous executions and to determine potential future improvements. The *planning* phase reconfigures the deployment plans, using the history-driven scaling propositions produced in the previous phase. The *execution* phase redeploys the cloud resources and the software tools for another execution. The *knowledge* about the workflows, and the cloud infrastructure (i.e., public/private providers, available types of resources, the existing resources, etc.) is a cross-cutting aspect serving information to each phase.

In the architecture of InfraComposer, the analysis and the planning phase are crucial. We employed a reactive [127] policy-based approach to adjust the deployment plans and resource allocation based on certain thresholds provided in policies. For example, a processing metric driven by statistics based on the execution history exceeds a particular threshold defined in the rules. This shows that the activity is a computationally intensive task. Then the configurator can reconfigure the deployment plan for that node to employ CPU-focused virtual machines.

Adaptation policies consist of rescaling rules. Each rule is composed of zero or more conditions, as well as a number of actions as a consequence. As depicted in Fig. 3.2, the metrics aggregator component executes complex statistical queries on time series data gathered during workflow executions, and it provides the aggregate data to the policy engine component through its interface. The scope of these reflective data includes cluster-wide metrics regarding the nodes (`Cluster`), the network (`Net`) and the storage systems (`Storage`), together with virtual-node related metrics concerning an average executing node (`VM`) and the coordinator node (`Master`). For instance, a condition can be `Net.bandwidth_utilization<1Gbps`. When a condition is satisfied, the policy engine performs rescaling actions (e.g. `resize`, `add`), which provides re-configuration hints to the configurator component. For instance, the resize action can be about nodes (e.g. number of cores, or the amount of memory), or it can be cluster-wide (e.g. number of nodes).

3.4.4 Monitoring Data Management

We first present more details about the data management components, and then we describe the scalability concerns and the possible existing solutions.

Monitoring data management components The monitoring data management of InfraComposer consists of (i) a set of telemetry data collectors (probes or agents) to inspect processes and system-wide metrics, (ii) data processing pipelines to parse incoming data streams, filter out irrelevant (or erroneous) entries, and transform them to the required format, (iii) a time-series database (TSDB) to store the execution history and (iv) a metric aggregator component including a data visualization layer to obtain aggregate inquiries, and eventually make them available for the policy engine.

For example, when a workflow engine dispatches jobs across a cluster of nodes, log entries regarding job events (e.g., started, completed, failed, etc.) are streamed through the monitoring components and stored in the TSDB. The

middleware is now capable of observing job count, job executions, throughput, and so forth. An example query can be the average percentage of maximum IO-wait during periodic time windows of 30 seconds for a specific python process, running on all worker nodes, between the start and the end of the previous round of the workflow execution.

Scalability of the stack The data processor component is perceived to be scalable depending on the monitoring stack. Telemetry data collectors (monitoring probes) should perform their tasks independently on worker nodes. The data processing pipeline can scale out horizontally since its functionalities are stateless and limited to parsing, filtering and transforming the input and eventually forwarding the stream to the time series database. Lastly, the chosen TSDB (e.g., InfluxDB, Elasticsearch, etc.) should be horizontally scalable. Recently, Jensen et al. [105] carried out a comprehensive survey about different angles of such databases including their architectural patterns.

Typically the data processing pipeline (e.g., Logstash) is the weakest link and it is likely to be CPU and network intensive [118]. More specifically, the processing functionalities include numerous CPU intensive executions of regular expressions through filtering plugins (e.g., Grok). In addition, it is network intensive, in the sense that if the input rate exceeds the maximum limit of the processing pipeline instance capacity, it will then throttle itself.

The first remedy to alleviate the workload is to perform preprocessing at the data collectors side. For example, data collectors should execute regular expressions, detect and filter unnecessary logs; in other words, they should lift the workload off the data processing pipeline before shipping the logs. Depending on the volume of workload, preprocessing is not always sufficient; in fact, the pipeline may again throttle itself against large-scale settings. The second additional solution is to employ messaging queues [118]. In this scenario, the processing pipeline instances need to get scaled out horizontally and follow a pull model against the queue (e.g., a Kafka topic).

3.5 Prototype Implementation and Use Case Validation

In this section we assess and validate our adaptive and reflective middleware for the cloudification of simulation workflows based on the use cases defined in Section 2. More specifically, we demonstrate a set of particular adaptive deployment scenarios and validate how both the reflective capabilities as well

as the adaptation capabilities of the middleware can cope with each adaptive deployment scenario. As a proof of concept, we implemented the concepts and architecture of InfraComposer in our prototype middleware and developed our two use cases (i.e., (i) the EWIS design and (ii) the design of the hinge system of an aircraft rudder (see Section 3.2)) as two simulation workflows that leverage the production workflow engine and actual simulation tools of the companies.

Simulation and optimization workflows, to achieve optimal objectives, go through an iterative execution of experiments. Each run of the experiments comes with different input design variables which are provided by the workflow engine at runtime. Input design variables are most of the time slightly different compared to the previous rounds. The behavior of a run is most likely predictable and more and less the same as the previous round. We can not be certain that these workflows are insensitive to the input data, however based on our observations, the behavior change is not very extreme. Therefore, in this work we assumed that every iteration of a workflow exhibits roughly the same behavior.

Moreover, we used OpenStack as a state-of-the-art cloud platform. Apart from the industrial domain tools in the use cases, the middleware is written in Java using the Spring framework, and the other technologies involved in each MAPE-K phase were: (1) Elastic Stack [118] as monitoring stack in the *monitoring phase*, including a set of telemetry data collectors (Filebeat and Metricbeat), a data processing pipeline (Logstash), a time series database (Elasticsearch) and a data visualisation layer (Kibana), (2) Drools [34] as rule engine in the *analysis phase*, (3) TOSCA [195] as orchestration and topology specification in the *planning phase*, and (4) Cloudify [75] as cloud orchestration and configuration management system in the *execution phase*.

In our validations, InfraComposer performs reconfigurations and adaptations through rescaling of cloud resources. We employed a policy-based approach for the analysis and the planning phase. We consider two styles of rescaling:

- *Vertical rescaling* supports to add more (or remove) resources to a single node in a system [63]. The middleware reconfigures resources to become larger or smaller resources. For example, a virtual machine with 2 CPU cores and 4 GB of RAM is reconfigured to 8 CPU cores and 8 GB of RAM.
- *Horizontal rescaling* supports to add more (or remove) nodes to a distributed system [63]. The middleware reconfigures the deployment plans to duplicate the nodes. For example, one experiment executes over 16 virtual machines instead of 4 virtual machines.

In the following sections, we revisit the use cases and validate the adaptation scenarios and monitoring results leveraging our adaptive and reflective

middleware.

3.5.1 Electrical Wiring Interconnection System Design

The simplified EWIS workflow (see Fig. 3.8) illustrates the simulation and optimization process of wire harness routing. Such workflow typically runs on a workflow engine (in our case Optimus [185]). To achieve an optimal wire routing, such optimization processes go through iterative execution of experiments in parallel. Each run of the experiments comes with different input design variables which are provided by the workflow engine at runtime. The workflow engine is responsible for the creation, coordination and offloading of the jobs to the worker nodes.

Following 3 different iterations (steps) in the MAPE-K loop, we performed several experiments presenting a number of adaptation scenarios. We assume that an engineer has made a set of inappropriate assumptions about the workflow and its requirements in each scenario. Each step is an iteration of the MAPE-K loop.

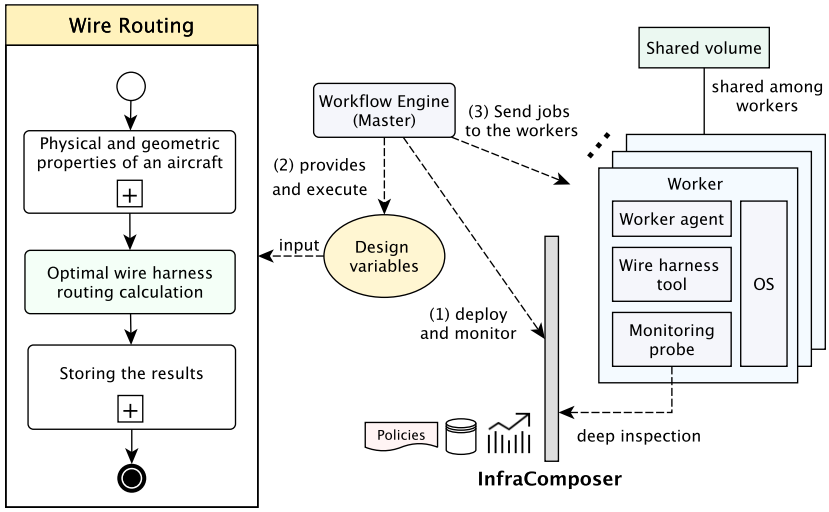


Figure 3.8: Overview of the workflow of wire harness routing simulation and optimization in the cloud. The workflow engine (master node) and the worker nodes are deployed in virtual machines.

	vCPU	RAM (GB)	Nodes	Intensiveness	Execution	Jobs per VM
Step 1	2	16	5	Memory Network	62.37 min	9
Step 2	4	8	10	CPU Network	33.10 min	∼5
Step 3	4	4	15	CPU Network	22.02 min	3

Table 3.2: Configuration details of each execution round of the EWIS workflow.

Scenario 1: the workflow is memory intensive The engineer annotates the wire harness tool as memory intensive, but the reflective monitoring shows that the tool does not load a very large chunk of data to the memory before and during the workflow execution. Consequently the deployment plan is altered to scale down and to employ virtual machines with a lower amount of memory.

In step 1, the workflow engineer specifies both direct deployment and resource consumption annotations. The former includes the use of the wire harness optimization tool, a single installation per node, a total of five worker nodes, and a shared storage for geometrical data. The latter describes the workflow as *memory* and network intensive. The workflow is then sent to InfraComposer to deploy the cloud infrastructure. As depicted in Fig.3.8, the execution then starts and the workflow engine (the master node) sends optimization jobs to the worker nodes. Given the above configuration, the total execution took 62.37min (see Table 3.2). In this table, the `Step1` settings are provided by the engineer, but the other steps are reconfigured by InfraComposer based on the adaptation scenarios and policies. Each execution step is an iteration of the MAPE-K loop which contains 45 experiments. Each node can take a job (optimal wire harness routing calculation) at a time in this use case. The number of experiments is specific domain knowledge, and InfraComposer observes the 45 experiments as one large experiment.

Afterwards, the engineer aims to re-execute the workflow with more experiments. InfraComposer monitored the execution of step 1, and it persisted the monitoring data after deep inspection of different processes, jobs, and system-wide metrics.

The monitoring component aggregates memory-related metrics (e.g., free, swap memory, etc.), and it transfers the results to the policy engine in order to enforce the business rules (see Table 3.3). As shown in Fig. 3.9, a large portion of the memory remained unused in step 1. Therefore, the virtual machine has been resized to employ less memory in order to avoid over provisioning.

S	Adaptations	Policies
1	– Assumption: memory intensive – Adaptation: scale up/down the VM	**input :** VM **if** *VM.free_memory* > 4*GB* *and VM.total_memory* >= 4*GB* **then** | resize(VM, VM.total_memory/2) **if** *VM.swap_memory* > 0 **then** | resize(VM, VM.total_memory*2)
2	– Assumption: CPU intensive – Adaptation: scale up/down the VM	**input :** VM **if** *VM.load_average* < *VM.cores* *and VM.CPU_utilization* < 40% *and VM.cores* >= 4 **then** | resize(VM, VM.cores/2) **if** *VM.load_average* > *VM.cores* *or VM.CPU_utilization* > 60% **then** | resize(VM, VM.cores*2)
3	– Assumption: few number of nodes – Adaptation: scale in/out the VMs	**input :** Cluster, VM, Workflow, Net, Master, license, quota **if** *(Workflow.execution* > 30*min* *or VM.accepted jobs* > 1*)* *and Net.bandwidth_utilization* < 1*Gbps* *and Net.packet_loss* < 5% *and is_sufficient (license, quota)* *and not master_node_saturated(Master)* **then** | quantity ← license.tools < 5 ? license.tools : 5 | resize(Cluster, Cluster.VMs.count + quantity) **if** *Net.bandwidth_utilization* >= 1*Gbps* *or Net.packet_loss* >= 5% *or master_node_saturated(Master)* **then** | resize(Cluster, Cluster.VMs.count - 5)

Table 3.3: Policy-based adaptation scenarios based on execution history. The system performance analysis is performed using the Utilization Saturation and Errors (USE) [82] methodology. Aggregated values are maximum values, and `VMs.load_average` maps to the system load during the last one-minute periods in Linux.

Scenario 2: the workflow is not CPU intensive The workflow (i.e., the wire harness optimization task in particular) is not annotated to be compute-bound, but the reflective monitoring data shows that the nodes and the tools utilize more than a particular threshold. More specifically, the virtual machine load average depicts system saturation by having a load number higher than the number of cores. Therefore, the deployment plan is updated to scale up and employ more cores.

To re-execute the workflow after step 1, InfraComposer enforces some compute-related rules using its policy engine (see scenario 2 in Table 3.3). Fig. 3.10 illustrates the CPU utilization as well as the system load during the last one-minute periods (Linux load average 1min). The Linux system load indicates system saturation and it should normally be less than the number of cores. In step 1, the virtual machine employed 2 cores and accordingly the system is saturated. Therefore the workflow is reconfigured to employ 4 cores instead, both for step 2 and step 3. As a result, the system load remained below 4 (the number of cores).

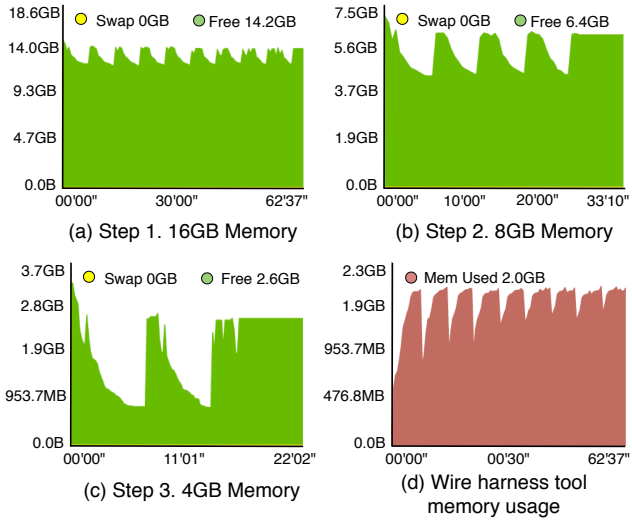


Figure 3.9: Different steps of the EWIS case: free memory vs. swapping

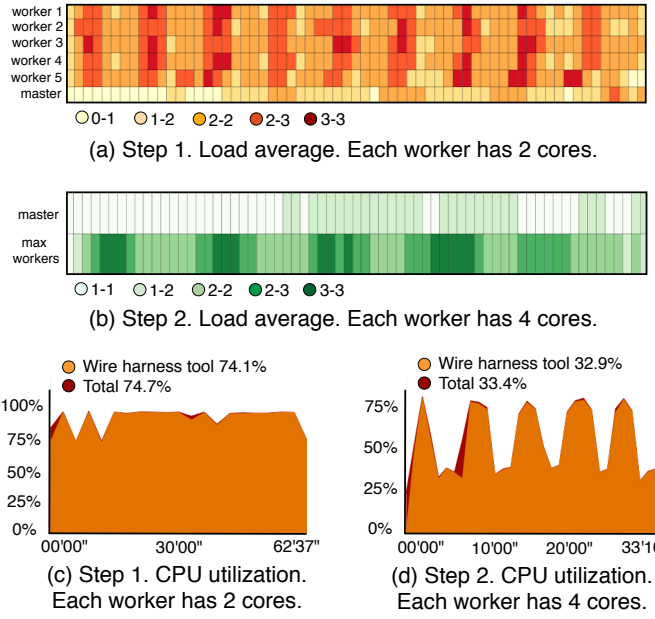


Figure 3.10: CPU utilization and load average of the workers and the master node. Master node has 4 cores. Each core is a virtual core.

Scenario 3: the workflow needs few number of nodes The engineer anticipates that the horizontal scale of the wire harness virtual machines should

be limited. He has to comply with his quota of total number of cores and memory in the distributed setup, and initially wants it to be deployed cheaply on a few number of instances. However, the reflective monitoring shows that there are a considerable number of parallel runs (i.e., scheduled jobs) due to the large size of input parameters and datasets (about the physical features of the aircraft). This results in a long-running overall execution. Consequently, the deployment plan is reconfigured to scale out the nodes in number and scale them down in memory size. As such the workflow can employ more instances of the tools, and can thus satisfy the expected execution time requirement while respecting quota and budget.

Policies in Table 3.3 for scenario 3 present horizontal re-scaling constraints as rules. In addition to execution time and number of accepted jobs per VM (9 jobs per VM in step 1), networking metrics, licensing and quota limitations are important too. Cluster-wide bandwidth utilization has an actual upper limit, and reaching that limit causes network saturation in terms of packet loss, networking errors and segments retransmissions. Furthermore, there should be available resources in the master node due to the execution of the workflow and the job scheduling. Policies make sure that the master node is not saturated in terms of average load, available memory, and disk IO rates.

(a) Step 2. Bandwidth utilization (max: 54.4 MB/s)

(b) Step 3. Bandwidth utilization (max: 84 MB/s)

Figure 3.11: Cluster-wide bandwidth utilization of the network where the shared volume is operating.

Fig. 3.11 illustrates the cluster-wide bandwidth utilization of the network where the shared volume is operating. The maximum inbound rate (34.9 MB/s in step 1 (not shown), 54.5 MB/s in step 2 and 84 MB/s in step 3) is lower than the overall bandwidth of the network, and therefore packet loss was negligible. In addition, disk IO utilization of the nodes were 52.4% in step 1, 74.82% in step 2 and 70.05% in step 3. The network latency of the shared volume has impact on these values. Therefore the policy engine scaled out and reconfigured the number of nodes to 10 in step 2 and 15 in step 3, and accordingly the execution time was 29.27min faster in step 2 and 40.25min faster in step 3 in comparison with step 1.

3.5.2 Design of the Hinge System of an Aircraft Rudder

As depicted in Fig. 3.12, this workflow employs seven engineering tools (denoted as `Tool1` to `Tool7`). These tools are responsible for meshing and stress analysis of different hinge components. Similar to the previous use case to obtain an optimized result (see Section 3.5.1), the workflow goes through iterative executions with different input design variables.

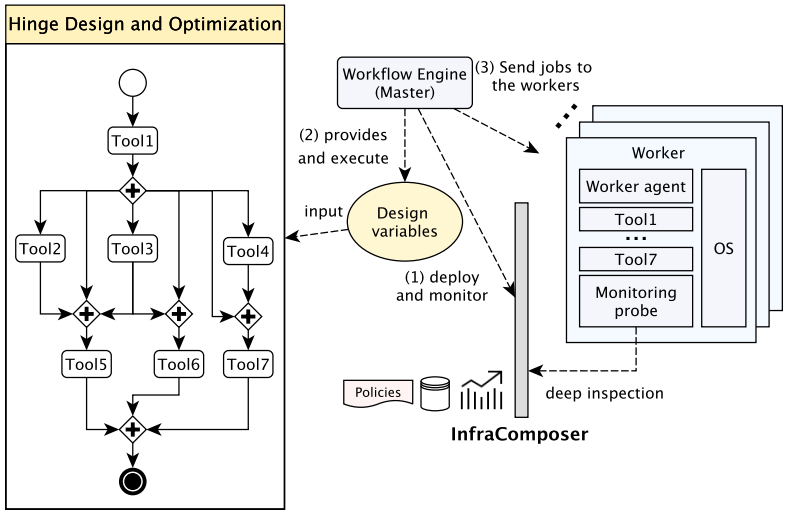


Figure 3.12: Overview of the execution architecture of aircrafts hinge system design and optimisation in the cloud.

	vCPU	RAM (GB)	Nodes	Intensiveness	Execution	Jobs per VM
Step 1	8	16	1	Memory CPU	109.20 min	315
Step 2	4	8	5	Disk	22.23 min	63
Step 3	2	4	10	Disk	12.12 min	∼32
Step 4	2	4	14	Disk	9.75 min	∼23

Table 3.4: Configuration details of each execution round of the hinge system design and optimization workflow. The `Step1` settings are provided by the engineer, but the other steps are reconfigured by InfraComposer based on the adaptation scenarios and policies. Each execution step contains 45 experiments.

The overview of the different experiments and results results are listed in Table 3.4. In step 1, the tools are annotated to be collocated on a single, large, powerful node, and the entire workflow is presumed memory and CPU intensive.

Scenario 1: the workflow is memory intensive The engineer annotates the entire workflow as memory intensive, but the reflective monitoring shows overprovisioning of resources. As illustrated in Fig. 3.13, the policy engine of InfraComposer scales down the deployment plan from 16 GB memory in step 1 to 4 GB memory in Step 3. As a result, the execution has a smaller yet more cost-effective footprint. Policies are listed in scenario 1 of Table 3.3.

Figure 3.13: The Hinge System case: average free memory vs. memory swapping in different steps.

Scenario 2: the workflow is CPU intensive The engineer annotates the workflow to be compute-bound, but the monitoring system shows that the compute resources are underutilized. In step 1, the deployment plan is set to employ 8 virtual cores, which results in maximum utilization of 7.2% for 315 sequential jobs (see Fig. 3.14). For re-execution of the workflow, the policies of the second scenario (see Table 3.3) trigger the InfraComposer configurator to scale down from 8 cores in step 1 to 2 cores in step 3. However, the CPU still remains unsaturated[1].

[1]Since the workflow has Windows-based worker nodes, there was no load average metric in place. The OS of the master node is Linux for both use cases.

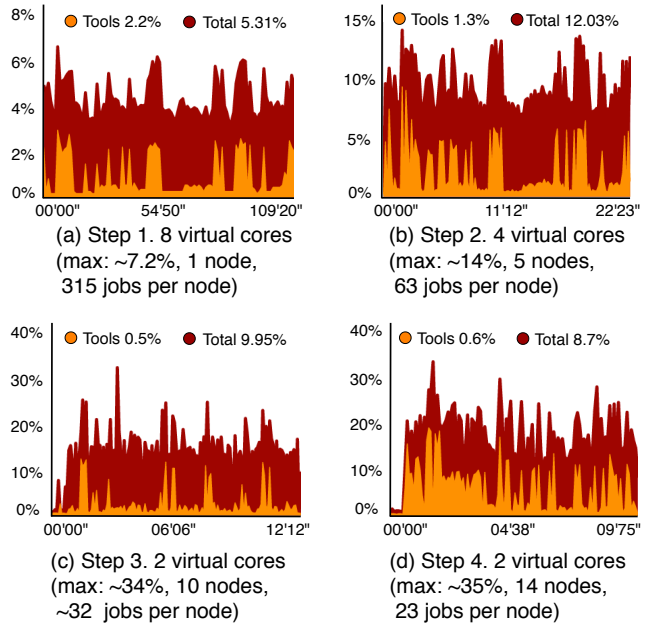


Figure 3.14: Maximum CPU utilization of the VMs and the tools.

Scenario 3: the workflow needs few number of nodes The engineer anticipates that a larger virtual machine is more efficient than a higher quantity of nodes. That results in an undesirable execution time of 109.20min. Beside other metrics, deep inspection of the workflow execution shows that this node received 315 jobs. Therefore, InfraComposer reconfigures the deployment plan to scale out horizontally and yet stay compliant with quota limitations when the engineer intends to re-execute the workflow.

As listed in Table 3.3, horizontal resource rescaling is constrained by cluster-wide metrics (e.g., overall bandwidth utilization) and master-node metrics (e.g., load average, free memory and disk I/O utilization). The network bandwidth utilization of the cluster has an acceptable rate of 7.8 KB/s in step 1, 39.1 KB/s in step 2, 293 KB/s in step 3, and 100 KB/s in step 4. As illustrated in Fig. 3.15, (a) and (b) represent the system load average during the last one-minute periods of the master node (i.e., with a maximum value of 1.4 in step 1 and 1.9 in step 4). Therefore, the master node remained unsaturated while managing 14 worker nodes. Furthermore, the master node never experienced memory paging, and it had free memory in all of the steps (i.e., with a maximum value of 6.4 GB in step 1, 6.1 GB in step 2, 5.8 GB in step 3 and 5.9 GB in step 4). Lastly, the Linux manual [76] defines disk I/O utilization as "percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for

the device). Device saturation occurs when this value is close to 100%". As depicted in Fig. 3.15, the percentage is around 12% in step 1 and 26% in step 4 at peak load moments.

As a result, the policy engine triggers the configurator component to resize the number of nodes to 5 in step 2, 10 in step 3 and 14 in step 4, and accordingly the execution time was reduced with 86.97min in step 2, 97.08min in step 3, and 99.45min in step 4. The distributed setup thus indeed scales well horizontally and the workflow executes much faster in comparison with step 1.

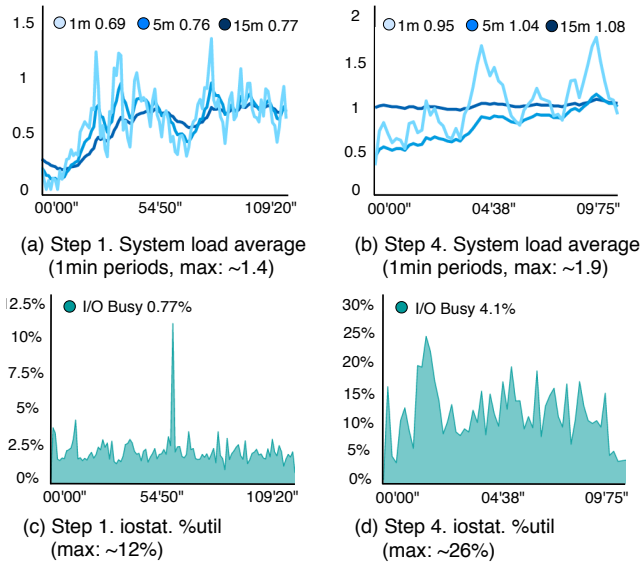


Figure 3.15: System load average and percentage of Disk I/O utilization of the master node which has 4 vCPUs and 8 GB RAM.

3.6 Limitations and Future Opportunities

In this section, we outline our design decisions and future research opportunities regarding (i) the granularity of componentization, and (ii) the policy-based decision making mechanism in the context of engineering workflow cloudification.

Granularity of Componentization InfraComposer is presented with virtual machines as the granularity of componentization; however, the concepts presented in the architecture are not necessarily tied to any specific component model (e.g., virtual machines or containers). Our future work includes applying

the concepts presented in this work to a containerised environment, which requires few considerations: (i) the reflective architecture and in particular the meta models (e.g., the resource meta model) should be adjusted to the new settings. (ii) The current architecture relies on a cloud orchestrator and a unified, modular, reusable deployment topology and orchestration specification such as TOSCA. Since mapping TOSCA to containers is not fully realised [60], there is a need for such an abstract specification enabling the integration with the state of practice container orchestration systems (e.g., Kubernetes, Mesos, Docker Swarm, etc.). (iii) Engineering workflows employ diverse set of domain tools executable on various operating systems; therefore, containers should be fully supported in those operating systems. (iv) Lastly a new autoscaling mechanism should be proposed in this context.

Dynamic Adaptation Policies The current InfraComposer architecture is based on a policy-based approach to define rules and in particular their thresholds (see Section 3.4.3, 3.5). This approach heavily relies on the domain knowledge of engineering workflows in order to prepare the policies, and on top of that, thresholds are statically defined. To dynamize the decision making machanism, Chen et al. [49] classifies the state of the art into (i) control theoretic approaches and (ii) search-based optimisation. The control theoretic approaches (e.g., Kalman control, Fuzzy control or PD) are well studied; however, they fall short of handling multi-objectivity (e.g., execution time, cost, etc.) and performing well where many rescaling decisions are supposed to be made.

On the contrary, search-based optimisation approaches such as reinforcement learning are promising based on the recent surveys [49, 127]. In this context, the complex nature of the engineering workflows cloudification and the vast optimisation space makes these approaches candidate to approach the problem. The optimisation space spans from single application configurations to co-location of services on nodes, processing components (VM or container), storage, networking, and so on. These tactics are appropriate because of well-studied related work in the domain of auto-scaling with regard to multi-objectivity through weighted sum, Pareto relation, and so forth, which is a requirement in our context.

Cloudification of engineering workflows requires a hybrid-level of control granularity (e.g., cloud resources and application level) entailing a large number of cloud primitives to monitor and tune in the adaptation process. Selecting relevant features to tune for the running workflow at hand is not an easy and trivial task to do manually. Therefore, an automatic detection of distinguishing abstract features in the execution history is an important improvement in this process (e.g., Chen et al. [48]; vPerfGuard[205]).

In summary, we envision the roadmap of a smarter deployment and execution of engineering workflows on the cloud within the MAPE-K control loop, which encompasses (i) exploring different granularity of componentization including containers and VMs; (ii) employing a smart, dynamic decision making approach such as search-based optimisation; (iii) detecting distinguishing features for the problem (workflow) at hand and their correlations with the given objectives; and lastly (iv) evaluating and validating the outcome in synthetic and most importantly real world cases and scenarios such as what has been achieved in this work regarding the policy-based approaches.

Threats to validity. We evaluated the middleware in an OpenStack cloud platform shared with other tenants. We have taken all possible isolation measures into account making sure that our performance numbers are not influenced by co-located tenants on the bare-metal compute nodes. Furthermore, the validated application cases are based on real-world aircraft designs, industry-standard tools and workflows. Due to the high financial cost of these tools and data, we had the opportunity of running our evaluations for a short window of time.

3.7 Conclusions

We introduced InfraComposer, a policy-driven, adaptive and reflective middleware, which enables simulation and optimization workflows to achieve smart and optimized deployment on cloud infrastructures. Our step-wise approach includes: (i) obtaining the engineers' input about initial, direct deployment of workflows through annotations in the first place, and (ii) the acquisition of new knowledge based on the actual execution history employed to produce improved deployments. Policies encapsulate knowledge of cloud engineers and system administrators to optimize the deployments based on execution histories. Such policies reason about reflective time-series data and act upon it by reconfiguring and resizing the execution environment for next iterations of the engineering workflow. As a validation and evaluation of the middleware, we presented specific adaptive deployment scenarios in real-life application cases in the domain of aeronautics. We validated how both the reflective and the adaptation capabilities of the middleware can cope with each scenario.

Chapter 4

DataBlinder: A distributed data protection middleware supporting search and computation on encrypted data

This chapter presents DataBlinder, a distributed data protection middleware supporting search and computation on encrypted data. DataBlinder provides crypto agility through configurable *fine-grained* data protection at the application level. DataBlinder supports adaptive selection of data protection tactics at runtime and offers a plugin architecture for such tactics based on a key abstraction model for protection level, performance, and supported query functionality. Under the hood, the middleware handles transparent distributed execution of the tactics by coordinating the different abstract steps in the tactics regarding secure indexing, distributed communication, and ensuring secure combinations of tactics without unexpected leakages. Lastly, to move computation closer to the data, we investigate the feasibility and usability of deploying cryptographic operations as user-defined functions (UDF) within widely used NoSQL database systems, namely Cassandra and MongoDB.

This chapter is based on an extended version of a publication at the 20th ACM/IFIP/USENIX International Middleware Conference Industrial Track

(Middleware Industry 2019) [17], as well as a publication in the Journal of Information Systems [168]. The content of this chapter is structured as follows. Section 4.1 introduces the domain of data protection, its latest advances, challenges and our contributions. Section 4.2 briefly provides a preliminary background on searchable encryption, property-preserving encryption, homomorphic encryption, and our application cases. In Section 4.4, we present two conceptual abstraction models for data protection tactics for security experts and protected data access for software developers, and afterwards, Section 4.5 describes the architecture and implementation of the middleware. Section 4.6 continues with a use-case validation and performance evaluation of the middleware. This section further investigates the feasibility and applicability of running cryptographic UDF within Cassandra and MongoDB. Lastly Section 4.3 and Section 4.7 conclude our research by providing the related work in this area and our concluding remarks.

4.1 Introduction

Software service providers use public cloud computing infrastructure to expand their computational capabilities and storage capacity. Outsourcing customer data to the public cloud is not always feasible for all domains of industry, especially the healthcare sector. The reason lies in the fact that a significant amount of their customer data is sensitive, and public cloud providers are considered to be *honest-but-curious*. As a result, service providers are reluctant to offload their sensitive data to external infrastructure. They either store their sensitive data on their premises or use data encryption to protect them against curious eyes.

Data protection regulations trigger companies to further enrich their security countermeasures to protect sensitive data, notably personally identifiable information. For instance, regulations oblige healthcare companies and organizations to notify the authorities regarding any data breach of *unsecured* protected health information (e.g., GDPR [169] and the HITECH Act [154, 153]). They are thus skeptical about employing cloud-based infrastructure and services, in particular, for storage of their critical data.

Healthcare providers are forced to deploy *data protection mechanisms* that go beyond encryption at rest or transmission. The state-of-practice data protection at rest using standard encryption is insufficient as software services are required to perform operations on encrypted data. They should be able to execute queries like:

- finding the patient with a particular gastric cancer who was admitted to the hospital in 12/05/2012 (boolean search),
- calculating the average heart rate of a patient (aggregate), or
- the number of times that the nurses refilled Doxycycline for a patient (aggregated search).

Latest advances. Researchers and practitioners have proposed many searchable encryption (SE) tactics and data protection systems for enabling search and computation on sensitive data in untrusted environments. Followed by Song et. al [186], a prominent body of research has been dedicated to symmetric SE (SSE). The research efforts have focused on defining security notions (e.g., IND-CKA2 [56]), building updatable and scalable schemes [43, 29, 30], optimal locality of encrypted indexes [45, 59, 43], and more complex functionalities such as boolean search [44, 109]. More practical, albeit less protective mechanisms, are based on property-preserving encryption (PPE), e.g., deterministic encryption (DET) [13], order-preserving encryption (OPE) [5, 25] and order-revealing encryption (ORE) [26] schemes. Furthermore, homomorphic encryption (HE) schemes allow us to operate, i.e., addition and/or multiplication, directly over encrypted data. HE schemes provide either addition or multiplication e.g., Paillier [159] and ElGamal [64]. Somewhat HE (SHE) and Fully HE (FHE) offer some combination of both at the cost of performance e.g., BGV [31] and TFHE [51].

Each of these tactics attempts to find a trade-off between *security*, *performance* and *functionality*. For example, encrypting the whole database (AES128) without searchability in mind provides us with a high degree of security but falls short of performance. Some tactics leak less information than others; among them, there are some with sub-linear search complexity. Each of these schemes offers different functionalities (e.g. equality, (con/dis)junction, etc.). Lastly, these advanced cryptographic constructions typically have complex designs leading to adoption difficulties by practitioners.

Challenges. There are several challenges for the development and application-level integration of such data protection tactics:

(i) there is no one-size-fits-all cryptographic scheme which maximizes all three aspects of the security, performance, and functionality trade-off;

(ii) integrating data protection tactics in the form of libraries to heterogeneous and polyglot software, e.g., microservice architectures, is prone to mistakes because such systems are developed using various ecosystems of programming languages;

(iii) the underpinning concepts and implementation details of cryptographic constructions used in data protection tactics are mostly complex for

application developers; in other words, choosing a right scheme as well as a secure and correct implementation are also prone to mistakes;

(iv) developing and incorporating new cryptographic schemes in an existing software stack is not a trivial and straightforward task for cryptographers; moreover,

(v) hypothetically, the computation of cryptographic functions (e.g., homomorphic addition) closer to the data within the database engine should be faster. However, the integration process in the scope of mainstream NoSQL databases is not straightforward and it is unclear whether this approach is feasible and practical—without changing the source code of the databases.

Contributions. We present DataBlinder, a distributed data-access middleware that encapsulates the complexity of data protection tactics. This middleware was developed in the context of an industrial applied research project [100] [1] in close collaboration with software service providers. It enables software service providers to seamlessly outsource sensitive data to the cloud-based services and yet be able to operate on it. The contributions are:

Adaptive selection of data protection tactics. We present an abstraction model to reify the data protection concepts, allowing application developers to request for their desired protection level and types of queries. The middleware accordingly selects appropriate tactics satisfying the requirements presented in the policies, and it adaptively loads the right implementation at runtime.

Extensible and pluggable architecture. Data protection tactics are subject to change to be more efficient, more secure and/or more expressive. Inspired by the comprehensive categorisation of Fuller et al. [71], we present an abstraction model for data protection tactics to reify their leakage profile, performance metrics and operations. As a result, tactic developers are provided with a set of interfaces based on the required operations using the Service Provider Interface (SPI) pattern, through which they plug in new tactics.

Recent research efforts have attempted to design and build secure database systems, such as CryptDB [166], Blind Seer [160], OSPIR-OXT [44, 43, 66], Arx [21], SisoSPIR [102], EncKV [207], etc. These systems employed different cryptographic constructions, such as SSE, various types of PPE, hardware-assisted approaches based on Trusted Execution Environments (TEEs), Oblivious RAM, and so on. The two key differentiating goals of our contributions in comparison to the prior research are (1) presenting an architecture enabling the software service providers to configure a notion of security with respect to

[1] https://www.imec-int.com/en/what-we-offer/research-portfolio/seclosed

their required operation, and (2) facilitating the current and future tactic extension process, unlike other systems that focused only on application developers and the cloud providers; therefore, our design is extensible and not tied to any specific tactic. More importantly, the adaptive and pluggable architecture take us one step closer to *crypto agility*, i.e., the ability to plug and play cryptographic schemes depending on their evolution in time.

To implement the architecture, some functionalities such as homomorphic encryption are required to run as close as possible to the data. Therefore, we implemented a dynamic deployment module for evaluating the feasibility, and applicability cryptographic functions within the NoSQL database engines in the form of User Defined Functions (UDF). For our investigations, we leveraged the UDF features of Cassandra and MongoDB and incorporated the homomorphic addition of the Paillier cryptosystem [159]. Our analysis concludes that the usability of this approach depends on the database in terms of their programming languages and systems support.

We validated the architecture by implementing several state-of-the-art data tactics leveraging our SPI's, and evaluated on FHIR-compliant [95] medical data. Our performance evaluations show that DataBlinder has limited impact on overall performance.

4.2 Background and application cases

This section briefly presents the cryptographic primitives and application cases.

4.2.1 Background

Over the past years, researchers and practitioners attempted to make practical SE constructions at the cost of allowing limited and defined information leakage, and yet retaining security in both the snapshot and persistent adversarial models. The snapshot model means the adversary obtains a snapshot of the secure index and the database, a well-motivated model for data breaches in the industry. The persistent model assumes that the adversary can observe all operations of the cloud server but without any interference.

Searchable encryption (SE). These schemes generally enable cloud providers to search for user-requested keywords on encrypted data without knowing the search word content and the plaintext data. These constructions are typically built on top of secure indexes that reveal no information (or formally defined leakage, also called leakage profile) about the content of search words and data

itself. In brief, they typically start with a setup protocol. This protocol generates the required keys, builds the initial index and prepares the cloud and local data stores. Next, the query protocol performs the search query by generating trapdoors (also called tokens) at the application side, which ideally reveal nothing about the search term. Using the trapdoors, the cloud provider can query the secure index by running an algorithm as a part of the search protocol and provide the querier with the encrypted document identifiers. Dynamic schemes include an update protocol for addition, deletion, and modification of the encrypted documents. *An example query can be searching for patients' details such as their health problems.*

Range queries on encrypted data. The main goal of such schemes is to allow cloud providers to compare ciphertexts without decryption by applying a *comparison* function. That enables the SE-based systems to build more complex queries such as range queries. Although the practical SE constructions built upon these primitives are recently subject to attacks [148, 84, 116, 83], they are still an ongoing research subject. *An example query can be searching for patients' health problems between particular date ranges.* Outside of the scope of this dissertation, there exist less practical but more secure constructions based on different cryptographic primitives such as fully homomorphic encryption (FHE) and Oblivious-RAM [78].

Homomorphic encryption (HE). Homomorphic encryption schemes encrypt data in a way that their underpinning mathematical properties enable the applications, in our setting the cloud providers, to perform certain operations on encrypted data such as addition or multiplication. Note that any function can be built as an arithmetic circuit, using solely addition and multiplication gates. The downside is that FHE or even SHE schemes that are capable to provide both, report poor performance in terms of computation and storage. Semi homomorphic schemes, however are considerably faster and have been commonly used in schemes that require arithmetic or geometric aggregation. In general, these schemes have been used in building aggregate functions in encrypted database systems. *An example query can be calculation of the average heart rates or body mass index (BMI).*

4.2.2 Application cases

Many businesses outsource various parts of their business activities to software-as-a-service (SaaS) providers, which use cloud computing platforms to run their software services. In most cases, customers' data is stored on the cloud platforms; however, their customers impose strict requirements on data protection and privacy. Therefore, SaaS providers are obliged to protect

sensitive data in a way that is secure against cloud providers and data breaches. Cryptographic protection techniques should at the same time preserve the required functionalities of these applications, such that SaaS providers can execute various queries like search or aggregations.

Electronic health record system. In healthcare, storage, and management of medical information is an important and critical aspect of the healthcare processes. Not only hospitals but also private practitioners require such software to manage patients' data. Medical data is one of the most sensitive types of data that requires special protection in software-defined systems. SaaS providers aim to offer electronic health record (EHR) systems to healthcare organisations. Therefore, putting strict cryptographic protections into practice is undoubtedly important and necessary. However, SaaS providers should be able to enable practitioners to perform various operations such as storage and retrieval of medical data. For example, in a medical laboratory, experiment observations (e.g. the amount of Glucose) are supposed to be stored and queried.

Invoice management system. In FinTech, many businesses outsource their billing services to the SaaS-based service providers, also known as billing as a service (BaaS). In such services, SaaS providers allow their customers to securely manage their financial documents and yet enable them to perform several different types of search operations. These cloud-based offerings make the recurring billing process as seamless as possible and provide a wide range of financial services, including the management of financial documents such as invoices, bills, accounts, and payments. The BaaS customers include organisations of all sizes from different application domains (e.g., banks, hospitals, telecom operators, etc.) with strict data protection policies. For example, one of the key operations is to search for invoices of a customer with search constraints such as paid or unpaid. Next to such search queries, SaaS providers typically run aggregate queries to provide the customers with statistical information (e.g. the average Internet fees for a region and the percentage of (un)paid invoices).

4.3 Related work

This section includes a state-of-the-art overview of searchable encryption (SE) and aggregate-based data protection tactics including searchable symmetric encryption (SSE), range queries and homomorphic encryption (HE). Furthermore, we present an overview of encrypted databases and middleware solutions that attempted to protect sensitive data in a pragmatic way. We outline how DataBlinder differs from state of the art.

Data protection tactics. Followed by the seminal searchable scheme of Song et. al [186], there has been a tremendous effort to offer practical constructions.

First, a large body of work have defined different security notions for SSE systems [28]. IND1-CKA [77] guarantees that no information about the content of documents can be learnt from the index. IND2-CKA further improved the definition by strengthening the security notion accordingly. Curtmola et al. [56] showed that protection of trapdoors are related to the search index, and as a result, they presented IND-CKA1 for non-adaptive security and IND-CKA2 for adaptive security. Both definitions determine that no information should be leaked from the index beyond search results and search patterns, and the trapdoors should not leak anything about search terms. Adaptive security notion means that an adversary can select its queries as a function of previous trapdoors and search results [56, 28]. IND-CKA1 and IND-CKA2 are the most widely used security definitions in this area. Bösch et al. [28] surveyed other definitions. The formal security definition of such settings was a challenge, which impacted our abstraction models.

Furthermore, research efforts have focused on building updatable and scalable schemes [43, 29, 30], optimal locality of encrypted indexes [45, 59, 43], and more complex functionalities such as boolean search [44, 109]. The slightly less secure–but more practical–line of tactics are based on PPE such as DET [13], OPE [5, 25] and ORE [26] schemes. Operations on encrypted data like addition and multiplication can be performed by HE [1] schemes, which includes partially HE (PHE) such as Paillier [159] and ElGamal [64], and fully HE (FHE) such as TFHE [51].

Encrypted databases. Designing protected search systems has been an active research area over the past years. CryptDB[166] is one of the seminal works, using onion of encryption that encrypt data in a layered approach for queries with different functionalities. The main goal is to keep the underlying legacy database unchanged. Pappas et al. [160] present Blindseer which is, unlike CryptDB, a custom database based on an approach using encrypted bloom filter trees as a storage mechanism. Fuhry et al. present the HardIDX [70] secure index system which leveraged Intel SGX to perform relatively efficient queries. Towards building secure NoSQL database systems, ARX[164] aims at presenting a protected system on top of MongoDB to provide the functionalities necessary to support associative arrays. EncKV [207] proposed a secure key-value store with a focus on secure and efficient partitioning of encrypted data and distributing data evenly across a cluster.

Database user-defined functions. Our aim is to bring the server-side (or cloud-side) operations of the protection tactics as close to the data as possible. Therefore, it is intuitively ideal to run the operations within databases. Using

user-defined functions (UDF) is one way to achieve this goal. CryptDB [166] presented most of its functionalities in MySQL as UDFs. Stankovski et al. [189] implemented Paillier in the Cassandra database and concluded that the execution time rises exponentially for large datasets, and most importantly, it is not ideal in situations where Cassandra clusters are composed of many nodes. Since this space has not yet been extensively explored, our research extends the research to document-oriented NoSQL databases, namely MongoDB.

Middleware solutions. Diallo et al. present CloudProtect [61], a middleware to enable users transparently encrypt sensitive data within various cloud applications. Their main goal is to support application functionalities while protecting sensitive data. CloudProtect uses deterministic encryption for search purposes, and on top of that, it has a policy-based protocol to expose sensitive data in plaintext for a limited duration on the server if some operation or function execution requires access to the data in plaintext. Alves et al. [128] present a framework for searching encrypted databases. They use ORE and HE for the range and aggregate queries.

There are several commercial encryption products such as Skyhigh Networks [149] and CipherCloud [53]. Most of these solutions employ legacy-friendly SE constructions to ensure that the existing applications can operate as before. Recently, Ionic [101] presented an encrypted search system with an advanced query construction mechanism based on EC-OPRF [36].

Most of these systems either proposed new SE constructions or employed fixed data protection tactics to provide their functionalities. However, the key goals of DataBlinder are to present a middleware solution allowing software developers to configure a notion of security with respect to their required operations at the application level, and it is not dependent on any particular database. Moreover, it facilitates the future tactic extensions via its architecture through SPI's.

4.4 Conceptual abstraction models

Our research approach considers two stakeholders: (1) software developers who aim to persist data in the database using regular operations in applications, and (2) security experts who want to secure these operations. In this section, we present two conceptual abstraction models accordingly: *the data protection tactic model* to abstract and reify different generic concepts found in most tactics, and *the data access model* to enable configurable tactic selection at run-time. In our research approach, we assume that sensitive data is in the form of a document that contains several fields. A database consists of several collections, and each collection is composed of many documents.

4.4.1 An abstraction model for data protection tactics

Each document is composed of fields. For example, a health record document may contain several fields such as a description, a sickness type and a numeric value indicating a measurable parameter like blood pressure. To protect sensitive values of the fields and yet support certain operations to clients, advanced data protection tactics are used. Each tactic typically offers a very limited number of operations, and as a result, a field employs multiple tactics to satisfy functional requirements of an application. As depicted in Fig. 4.1, a tactic has a set of internal operations. Each of these operations comes with a leakage profile and several performance metrics. Our abstraction model is inspired by the recent SoK paper of Fuller et al. [71] published in IEEE Security and Privacy 2017.

Tactic operations. Tactics include one or several operations [71]. In general, (i) the *init* operation to set up cryptographic primitives and initial provisioning of data structures and databases, (ii) the *update* operation for dynamic tactics to add, update and delete documents, and (iii) the *query* operations to perform tactic-specific functionalities such as boolean search. The *query* operation, illustrated in Fig. 4.1, can be various search or aggregate queries depending on the given protection tactic.

Leakage profile. Data protection tactics in such systems rely on secure data structures, e.g., an encrypted index, to facilitate data retrieval in an efficient way. These systems, notably their constitutive data structures, are susceptible of leaking (meta-)information. For example, a commonly used structure is an encrypted inverted index. Leakage could be something trivial such as the result size of every possible search word. There are a wide range of leakage profiles with different levels of severity. Encrypted indexes usually contain information about searchable keywords and document identifiers. Searchable keywords are derived from the content of the documents. Document identifiers uniquely point to the documents in a database.

To illustrate several types of leakages, we assume that, as an example, there are three tactics with their searchable data structures roughly similar to a reverse index model. In a database, documents have unique identifiers (id_1,...,id_6) and a set of words (w_1,...,w_5) as search terms extracted from the document contents.

In Table 4.1 for instance, searching for w_4 results in document id_2 and id_4, where F means a pseudorandom function (PRF) such as a keyed hash function like HMAC. Prior to any query, the construction leaks the document identifiers, the number of documents, the number of words, the size of documents, frequency of all words, and co-occurrence information for all words between documents. Upon issuing a search query, the search result (document identifiers) and equality pattern of the search terms (but not the value) are leaked to the server. The

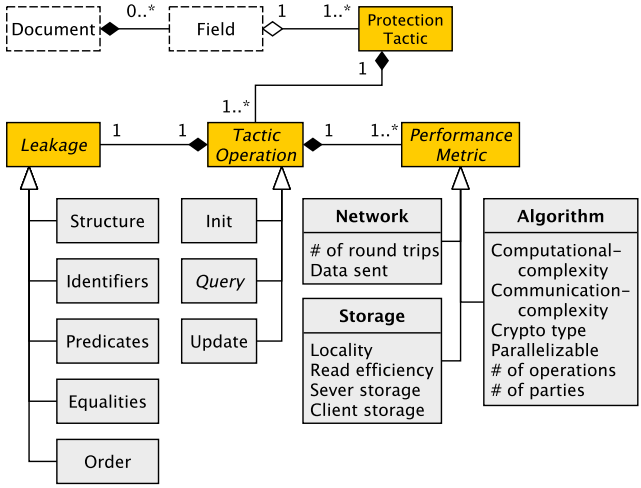


Figure 4.1: Abstraction model of data protection tactics for tactic providers

former and the latter are called access and search patterns. Upon update queries (addition or deletion), the server learns which document identifiers are being changed and whether these changes are reflected in the index and the search results in comparison with prior searches. This leakage is related to the backward and forward privacy [46].

words	identifiers
$F(w_5)$	id_3, id_1, id_5
$F(w_1)$	id_2
$F(w_4)$	id_2, id_4
$F(w_3)$	id_1, id_4, id_6

Table 4.1

words	identifiers
$F(w_5)$	$\overline{id_3}$, $\overline{id_1}$, $\overline{id_5}$
$F(w_1)$	$\overline{id_2}$
$F(w_4)$	$\overline{id_2}$, $\overline{id_4}$
$F(w_3)$	$\overline{id_1}$, $\overline{id_4}$, $\overline{id_6}$

Table 4.2

words	identifiers
$F(k_5, 1)$	$\overline{id_3}$
$F(k_5, 2)$	$\overline{id_1}$
$F(k_5, 3)$	$\overline{id_5}$
$F(k_1, 1)$	$\overline{id_2}$
$F(k_4, 1)$	$\overline{id_2}$
$F(k_4, 2)$	$\overline{id_4}$
$F(k_4, 1)$	$\overline{id_2}$
$F(k_4, 2)$	$\overline{id_4}$
$F(k_4, 3)$	$\overline{id_6}$

Table 4.3

Table 4.2 illustrates a further secured index by encrypting the search results (identifiers, denoted by $\overline{id_i}$), where the encryption function is a semantically secure symmetric encryption and encryption keys are derived from the words. This construction protects the value of identifiers, the size of documents as

well as co-occurrence information of the search terms between documents, but it still leaks the number of search words and their frequency. Moreover, the index allows the server to further learn the search and access pattern and it is not backward and forward private. Table 4.3 demonstrates a slightly different searchable encrypted data structure offered by a different protection tactic. In this construction, a set of cryptographic keys (k_1,...,k_5) are derived from the words (i.e. k_i is derived from w_i), and the table is built based on the keyed hash of separate counters for each word. To search for a term, the client needs to produce a derived key for the search term and provide the server with that key. Prior to any query, this scheme, unlike others, protects the number of words and the frequency of them. However, it also leaks the access and query patterns, and it is not forward and backward private in case of update queries.

Prior work presented various formal security definitions for searchable encryption [28, 77, 56], and other work categorised the leakage levels [42]. We employed the leakage taxonomy presented by Fuller et al. [71] for the reification of this concept due to its generality and applicability.

To present a middleware solution, having a generic classification of leakage profiles does not capture all specific cases for each operation. For example, the update operations (create, delete, and update) are sensitive. They might leak information about the future or the past. Certain leakages occur prior to any query in setup time (having a snapshot of the database) or at query time such as boolean queries. The pragmatic reification of data protection level in a data-access middleware motivates the idea of presenting the leakage profiles on a per-operation basis.

The leakage level of data protection tactics can be concretely categorised into five levels [71]:

(i) *structure*, i.e. nothing is leaked except the size of the entire data structure or things which can be hidden by padding,

(ii) *identifiers*, i.e. past and future access patterns of identifiers are leaked,

(iii) *predicates*, i.e. complex query predicates leak information such as intersection of a boolean query with a known range,

(iv) *equalities*, i.e. which objects have the same value in the system, and

(v) *order*, i.e. the numerical and lexicographic order of the objects are leaked.

The leakage level of *order* is the highest, and *structure* is the lowest (the most secure one).

Performance metrics. Each tactic operation also comes with a performance cost impacting clients' experience. As illustrated in Fig. 4.1, the performance of data protection tactics can be measured and quantified by certain types of metrics related to the underpinning algorithms, network, and storage overheads. Tactics can employ various algorithmic designs to securely execute operations, which in turn may affect performance differently, e.g., tree based search vs. exhaustive search. Those decisions consequently have impact on networking infrastructure in terms of data sent and received between clients and providers. Besides, tactics may have severe impact on the locality of objects, read efficiency, the size of data storage both at the client and the server side. For example, to improve security by a tactic, relevant data might not always reside in close proximity on a disk or a partition. Lastly, the type of cryptography has a considerable impact on computational resources. For instance, public-key cryptography and homomorphic encryption used by some tactics are significantly expensive in comparison to symmetric encryption. These metrics are mostly inter-related to each other.

4.4.2 An abstraction model for protected data access

Data protection tactics should be applied to documents with per-field granularity. Fig. 4.2 illustrates the data access abstraction model for sensitive fields, which primarily includes (i) which data-access and aggregate operations can be performed on a field, and (ii) up to which level sensitive fields are protected. This abstraction model is aimed for application developers.

Each field of a document can be annotated with the model illustrated in Fig. 4.2. For instance, consider a medical document containing the patient's age. We can select its sensitivity level, and assign a *Class 2* protection level (explained later). We can then configure what operations are needed, in this case *Average* and *Equality Search*. The middleware employs the appropriate implementations at runtime accordingly in order to meet the client's requirements.

Query functionality. A data access middleware should in general offer all required query functionalities to the client applications. Fuller et al. [71] present a set of base operations built upon relational algebra, associative arrays and linear algebra which are essentially the foundation of SQL, NoSQL, NewSQL and polystore systems. We employ this categorisation of base operations as query interfaces in order to satisfy the data-access requirements of many applications.

The core functionalities of the query interfaces rely on key operations of associated arrays [138, 80] and basic functions of persistent storage systems [134], which are namely (i) create, (ii) read, (iii) update, and (iv) delete. The read operation goes beyond fetching a document; it comprises more complex search

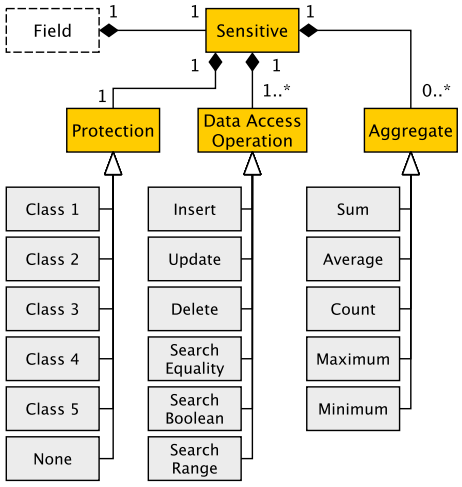


Figure 4.2: Data access model of the sensitive fields for application developers

operations with predicates specifying conditions such as (i) equality, (ii) boolean queries (conjunction, disjunction and negation), (iii) range, and (iv) others. These operations can be combined with aggregate functions such as sum, average, count, maximum, minimum, and so on. Each operation could be mapped to one or more data protection tactics.

Data protection level. Unlike the model of data protection tactics in which leakage profiles are specified per operation, the data access model specifies a protection level per field. In the previous model (Fig. 4.1), the reification of per-operation leakage facilitates the extensibility of the middleware architecture. However, in the data access model (Fig. 4.2), it is meaningless to concretise a per-operation protection level for each field. For example in a medical document case, the patient's age is set as sensitive, and the equality and the range-search are set as the required functionalities. If two different data protection tactics are employed by the middleware to satisfy the operational requirements, the one which supports range queries leaks more information compared to the other one. In other words, the protection level of a field is equal to the tactic with the weakest guarantees regardless of the strength of other tactics applied to it (i.e. a chain is only as strong as its weakest link.). *We therefore conclude that a data-protection level per field is appropriate for the data access model.*

We classify data protection tactics into five classes (Class_1,...,Class_5) of protection guarantees. Each of these classes corresponds to its counterpart in the data protection model. Class_1 has the least leakage. However, the model is sufficiently open to be realised differently; the selection of protection tactics

has to be controlled through security policies.

4.5 Architecture and implementation

In this section, we present an overview of the DataBlinder middleware architecture, and we further describe the extensibility and pluggability of the architecture by introducing tactic commonalities and service provider interfaces (SPI).

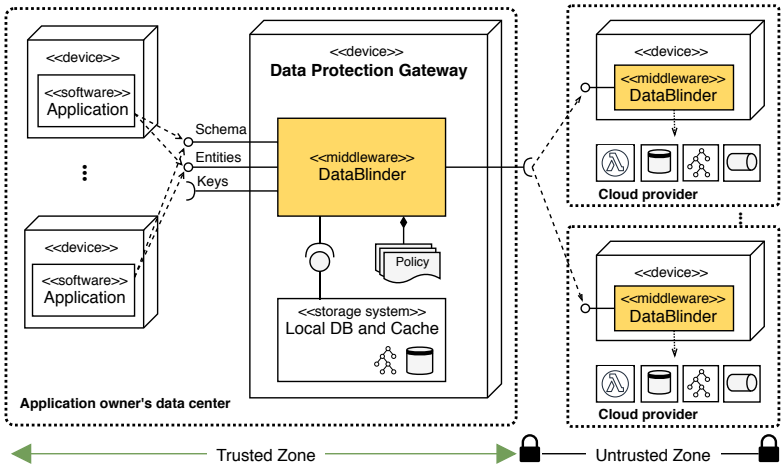


Figure 4.3: Middleware deployment view

Deployment topology and system overview. The setting consists of a trusted zone, which is the application owner's datacenter, and an untrusted zone composed of external cloud providers (see Fig. 4.3). In the trusted zone, different types of applications can benefit from external cloud-based storage through a data protection gateway. The current design of DataBlinder is based on the microservice architecture. That means DataBlinder acts as a gateway, a standalone service, to serve various types of other services within the on-premise data centre of application service providers. The DataBlinder middleware along with its required assets such as data protection policies and storage facilities are deployed within the gateway. There are several interfaces exposed to the applications, which are namely a *Schema* interface to enable clients to define and annotate data schemas and data protection metadata, an *Entities* interface to allow regular data-access operations, and a *Keys* interface to allow the system to integrate with on-premise key management systems (e.g., HSM). All data-access

operations are trusted, and the inter-application communications within the datacenter follow regular security and access control mechanisms.

The untrusted zone consists of several cloud providers and the communication channels between the application owner's datacenter and these external resources. The middleware is distributed since SE tactics are inherently distributed.

4.5.1 The middleware architecture

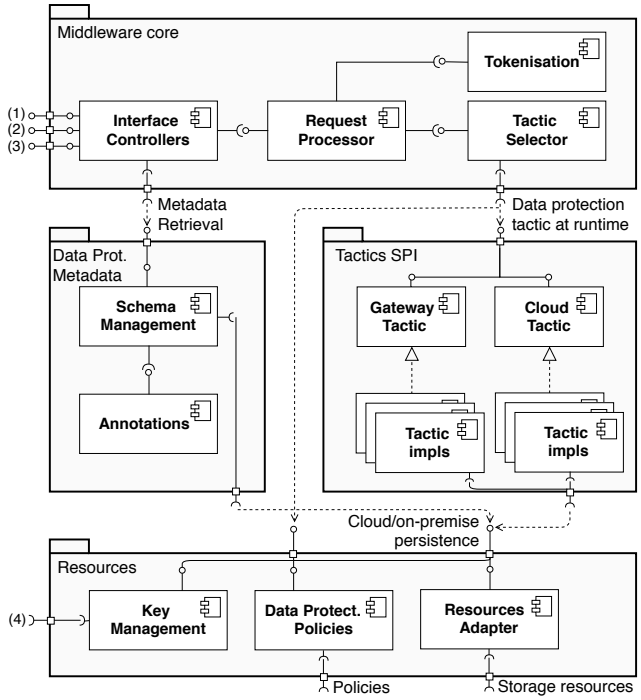


Figure 4.4: Middleware component diagram.

Fig. 4.4 illustrates four subsystems of DataBlinder. Depending on the deployment location, either in the trusted or the untrusted zone, different interfaces and components are employed. (i) *The middleware-core subsystem* is responsible for the abstract execution of the persistence logic, e.g., Create Read Update Delete (CRUD) operations, and adaptive and dynamic tactic selection at run-time. (ii) *The data protection metadata subsystem* is responsible for the persistence and retrieval of per-application database schemas and data protection annotations. The schema management component also validates whether the

application documents correspond to the configured schemas. (iii) *The tactics SPI subsystem* is responsible for providing concrete tactic implementations, comprising a set of gateway and cloud implementations. (iv) *The resources subsystem* is responsible for enabling the access to external resources such as cryptographic key management systems, and on-premise or cloud-based resources such as storage systems.

4.5.2 Data protection metadata specification and verification

In this section we present (1) how document schemas and data protection annotations are managed in a per-application basis; and (2) how we concretely employ and extend the JSON-Schema specification [204] for documents towards abstraction models (see Section 4.4.2).

Schemas & metadata per application. DataBlinder manages document schemas and data protection annotations per application. In brief, the middleware can serve a set of applications. Each application is composed of a number of databases, and each database is a set of collections. In document-oriented databases, collections are akin to tables in relational database systems. The schema c of a collection describes the data model of that collection, which is concretely based on JSON-Schema specification and our extensions. For example, in a medical application composed of a database for medical history, a lab measurement is a collection. Data models of collections are managed per application with per-collection granularity.

JSON-Schema with data protection annotations. DataBlinder adds the notion of data protection per field to the JSON-Schema specification. JSON-Schema is a document which describes the structure of a JSON document [204]. Each schema document defines a set of properties annotated with different information such as type, whether a property is required, maximum value, and so on. At runtime, the instances of documents get validated against the schema document.

JSON-Schema is solely responsible for describing the JSON document structure, and offers a means of validation. To add the notion of data protection to the JSON-Schema in the context of DataBlinder, our contributions are twofold:

- *Sensitivity annotation.* In Section 4.4.2, a data access abstraction model is presented. To add the data protection notion, we introduce a new annotation called *sensitive*. The keywords and the structure of these

annotations will be per property and based on the presented model in Fig. 4.2. The details are explained later in this section.

- *Schema validation and verification at runtime.* Once properties of JSON documents are annotated with sensitives fields via JSON-Schema, the JSON-Schema Validation process should be extended in a way that the validation process can be performed correctly and take these annotations into consideration at runtime (syntactically and semantically).

To present our extensions more formally, the syntax of the data protection annotations are provided below with the extended Backus-Naur form (EBNF) [103]. For readability, terminals are written in bold.

<*sensitive*> ::= **sensitive : {**<*pairs*>**}**
<*pairs*> ::= <*protection*>**,**<*operations*>**,**<*aggregates*>
<*protection*> ::= **protection :**<*protClass*>
<*protClass*> ::= **class1** | **class2** | **class3** | **class4** | **class5** | **none**
<*operations*> ::= **operations :** [<*operation*>(**,**<*operation*>) ∗]
<*operation*> ::= **insert** | **update** | **delete** | **search_eq** | **search_bool**
<*aggregates*> ::= **aggregates:** [<*aggregate*>(**,**<*aggregate*>) ∗]
<*aggregate*> ::= **sum** | **avg** | **min** | **max** | **count**

Pezoa et al. [163] present a complete formal grammar of JSON-Schema. The above listing describes the syntactic definition of our extensions.

Semantic validation. The metadata component should be capable of *semantic* validation of data protection annotations. The standard [103] defines six types of properties: strings, numbers, boolean, objects, arrays and null. Annotating a sensitive property to use a tactic should be semantically feasible; for example, the combination of a boolean type with an aggregate function based on an additive homomorphic encryption scheme does not have any meaning. The schema management component controls these aspects. However, no thorough validation is covered in this dissertation since it requires a deep investigation of all corner cases. Therefore, this angle of the middleware is left for future work.

4.5.3 Extensible and pluggable architecture

In this section, we present a more concrete description of the extensibility and pluggability of the DataBlinder architecture by introducing the practical aspects of tactic commonalities, and tactic selection at runtime.

An extensible software architecture is the one embracing the future changes, where its components can be extended without (or minimum) modification of the core functionalities. Weck [200] introduces an extensible component architecture with a set of properties: (1) users can compose the components at runtime; (2) components are developed by different people; (3) components should work with or beside each other (or at least with no interference); and, (4) components should be extensible in the directions which have been anticipated and prepared for (also called dimension of extension). Such properties can be satisfied with design rules for component developers, typically leading to certain abstractions. In Section 4.4, we presented two abstraction models to reify certain concepts about protection level, performance and query expressiveness. The models enable the application developers to compose their required tactics, and the tactic developers to define their constructions (see Section 4.5.2).

Tactic commonalities. Most tactics share common properties. They are all distributed in the sense that two or more parties are involved in performing a high-level operation such as boolean search. The comprehensive surveys on SE [165, 28, 71] distill their life cycle into three key operations: *setup* for key material generation and initial index provisioning, *update* for dynamic constructions with the operations like deletion, addition and modification, and *query* for constructing tokens and performing the given function.

Each data protection tactic includes a subset of operations. Each of these operations is a distributed protocol. As a result, tactics share a common framework in the DataBlinder architecture supporting: (1) gateway and cloud implementations per operation, (2) cryptographic primitives as building blocks (e.g., PRF), (3) key management integration, (4) communication channels for transferring protocol data, and lastly (5) data repository services available to both the gateway and the cloud implementations to satisfy tactic-specific requirements to construct distributed secure indexes.

Tactic SPI. The DataBlinder protection tactics can be extended by leveraging a set of interfaces. Each of these interfaces exposes a high-level operation defined in the data-access abstraction (see Fig. 4.2), including a gateway and a cloud version as described earlier in this section and Fig. 4.4. The first interface which is mandatory to implement for all tactics is the *setup* interface. The other major but optional operations include CRUD, various search and aggregate queries. Each implementation receives all dependencies required to perform its protocol as listed in the commonalities. Table 4.4 lists the interfaces for a large subset of the high-level operations.

	Gateway Interfaces	Cloud Interfaces
Insert	Insertion, DocIDGen, SecureEnc	Insertion
Update	Update, DocIDGen, Retrieval SecureEnc	Update, Retrieval
Delete	Deletion	Deletion
Read	Retrieval, SecureEnc	Retrieval
Equality Search	EqQuery, EqResolution <*Read*>	EqQuery
Boolean Search	BoolQuery, BoolResolution <*Read*>	BoolQuery
Aggregate	<*Query*>, AggFunctionResolution	AggFunction

Table 4.4: **Service Provider Interface (SPI)**. The implementations of these interfaces get loaded dynamically at runtime. <*Read*> and <*Query*> denote a set of interfaces required for a retrieval and a search operation.

Tactic selection at runtime. The SPIs are implemented by security experts. The middleware loads the right implementations dynamically at runtime using the strategy design pattern [74]. Fig. 4.5 and Fig. 4.6 illustrate a high-level description of the setup and the search protocol, where `S` is a schema; `Pr` is a search predicate; `P` is a data protection policy; `D` is a set of decisions made by the policy engine; $\texttt{PL}_\texttt{g}$ and $\texttt{PL}_\texttt{c}$ are gateway and cloud payloads; `ch` is a communication channel; `STs` is a list of search tokens; `ctx` is the context; `EIDs` are encrypted document identifiers; `IDs` are plaintext document identifiers; and *SO*, *QO* and *RO* are setup operation, query operation and resolution operation tactics which are dynamically loaded based on the decisions and the context. Lastly, ***`args`*** are the required services described in the commonalities section provided by the middleware such as key management service, and cryptographic building blocks, and so forth.

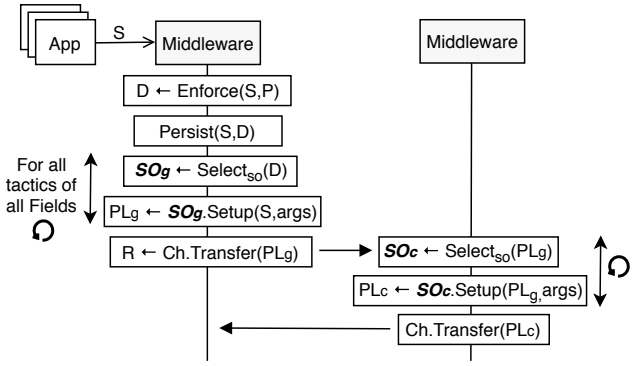


Figure 4.5: Setup protocol

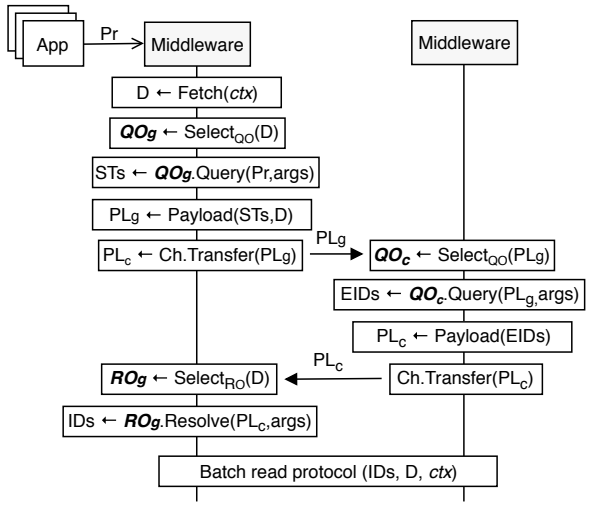


Figure 4.6: Search protocol

In the setup protocol, a document schema with sensitivity annotations is submitted to the middleware. First, the data protection policies are enforced to select which tactics are appropriate for which field for the given application requirements (see Fig. 4.7), and the end result is a list of decisions `D`. In fact, the characteristics of each tactic are reflected by the rules of the policies; therefore, the actual tactic selection based on the abstraction models happens at this stage. Consequently, an implementation for the setup operation can get dynamically loaded at runtime based on `D`. Afterwards, the setup implementation is executed by the middleware and the payload is transferred to the cloud. As a result, the same set of tactics are selected with no policy enforcement overhead and the protocol continues the execution.

In the search protocol, applications issue queries by sending a search clause containing predicates (`Pr`). Search protocols, and query protocols in general, employ decisions (`D`) made already in the setup phase. The right search tactic is loaded similar to the previous protocol, and a set of search tokens (`STs`) are generated as a result. A normal secure search tokens should not leak any information about the search keywords (e.g. based on IND-CKA2 [56]). These tokens are transferred to the cloud-side in a payload ($\texttt{PL}_\texttt{g}$) through the channel. Likewise, the cloud-side query operation is loaded and executed. The typical implementation should query the secure index using `STs` and fetch the encrypted document identifiers (`EIDs`). The result is sent back to the gateway side, and a resolution operation deciphers the document identifiers.

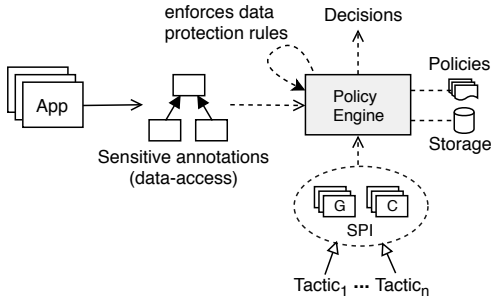


Figure 4.7: Tactic selection based on sensitivity annotations and data protection rules. Dashed lines mean execution flow within the middleware. `G` and `C` are gateway and cloud-side interfaces.

4.6 Use case validation and evaluation

To validate and evaluate the concepts and architecture of DataBlinder, we seek to answer:

Q1 How broad and extensible is the architecture, in particular SPI's, to support the state of the art data protection tactics?

Q2 What impact on performance does the DataBlinder architecture have?

Q3 What is the impact of running cryptographic code within User Defined Functions (UDF) inside NoSQL database engines?

4.6.1 Proof of concept development

DataBlinder supports two modes of execution: *gateway* and *cloud*, implemented using Spring Boot, i.e., ~ 6,000 lines of Java 8. We employed libraries such as Bouncy Castle for basic cryptographic primitives (e.g., AES/GCM, RSA/OAEP, HMAC-SHA256, etc.). We further leveraged the Clusion [115] project to provide several data protection tactics,and Javallier [193] for the Paillier [159] cryptosystem. The DataBlinder data protection tactics have been developed using these building blocks. We employed document-oriented databases, e.g., MongoDB and Elasticsearch, to store documents and indexes. We also employed a key-value datastore, e.g., Redis, in a semi-persistent durability mode to take advantage of basic constructions such as persistent sets, maps, and so on, to build custom indexes.

		Protection level		SPI*			
Operation	Scheme	C#	Leakage	G	C	Challenge	Impl.
Equality Search	DET	4	Equalities	9	6	-	○
	Mitra [46]	2	Identifiers	7	5	Local storage	○
	Sophos [29]	2	Identifiers	6	4	Key management	○
	RND	1	Structure	6	4	Inefficiency	○
Boolean Search	BIEX-2Lev [109]	3	Predicate	8	5	Storage impl. complexity	[115] ◗
	BIEX-ZMF [109]	3	Predicate	8	5	Storage impl. complexity	[115] ◗
Range Query	OPE [25]	5	Order	3	3		[131] ◗
	ORE [26]	5	Order	3	3		[119] ◗
Sum	Paillier [159]	-	-	3	3	Key management	[193] ◗
Average	Paillier [159]	-	-	3	3	Key management	[193] ◗

Table 4.5: These cryptographic constructions have been implemented and integrated to DataBlinder using the tactic interfaces. $*$ denotes the number of service interfaces required in the implementation (Table 4.4); ○ denotes that we implemented the construction; ◗ denotes that the implementation is slightly modified; C# is a class number; C and G are the abbreviations of Cloud and Gateway.

Tactic integration. We implemented and integrated several tactics using the proposed achitecture based on the SPI pattern (see Table 4.5). The implementation covers a broad range of tactics, having various properties, such as different protection levels, forward privacy (e.g. Mitra and Sophos), deterministic and probabilistic encryption (e.g. DET and RND), read and space efficiency (e.g. BIEX-2Lev and BIEX-ZMF), data order (e.g. ORE and OPE), and HE (e.g. Paillier).

4.6.2 Healthcare use case

To validate the middleware, we present a real-world example of the industry-standard FHIR-compliant [95] medical documents. We annotate the schema based on some assumptions regarding protection level and functionalities.

Example. Observations are measurements and assertions about patients [95]. In the following document, the amount of Glucose observed in a blood test is illustrated. Most of these fields are assumed to be sensitive since they can be the indicators of diabetes.

```
{
 id: f001,
 identifier: 6323,
 status: final,
 code: Glucose,
 subject: John Doe,
 effective: 1359966610
 issued: 1362407410,
 performer: John Smith,
 value: 6.3,
 interpretation: High
}
```

Sensitives	Annotations
status	C3, op [I, EQ, BL]
code	C3, op [I, EQ, BL]
subject	C2, op [I, EQ]
effective	C5, op [I, EQ, BL, RG]
issued	C5, op [I, EQ, BL, RG]
performer	C1, op [I]
value	C3, op [I, EQ, BL], agg [avg]

Sensitives	Tactic Selection	Reasons
status	BIEX-2Lev	Boolean & cross-field
code	BIEX-2Lev	Boolean & cross-field
subject	Mitra	Identifier protection level
effective	DET, OPE	Range queries
issued	DET, OPE	Range queries
performer	RND	Structure protection level
value	BIEX-2Lev, Paillier	Cloud-side averages

C is a class; op is a list of operations; I,EQ, BL and RG are insertion, equality, boolean and range queries; agg is the list of aggregate functions; and avg is an average operation. DataBlinder enforces data protection policies to perform tactic selection, and it abstracts away the complexity of underpinning cryptographic protocols. Therefore, software developers only require the necessary knowledge about the data-access abstraction model. Moreover, the middleware decouples the applications built on top, in the sense that the evolution of the tactics has limited impact on the applications with respect to the functionality and data protection requirements.

4.6.3 Performance evaluation

We evaluate the overall performance overhead of DataBlinder in comparison to the scenarios where: the application only does data operations and does not use the middleware or any tactic (S_A); the data protection tactics are implemented hard-coded into the application without using the middleware (S_B); and the application uses DataBlinder to enforce the required data protection tactics (S_C).

Set-up. To evaluate the performance overhead of DataBlinder, we developed the middleware as presented in Section 4.6.1, and we deployed an instance of it on the Openstack private cloud in *gateway mode* and another instance on

a public cloud provider (Microsoft Azure) in *cloud mode*. Our underpinning Openstack compute node comes with 2.60 GHz Intel Xeon E5-2660 processors and 128GB DDR3 memory. The gateway VM instance has 8 vCPU cores and 16GB of RAM. The cloud VM instance has 4 vCPU cores and 16GB RAM. We deployed an instance of Redis in a semi-durability mode on both sides and an instance of MongoDB on the cloud. We deployed an instance of Locust [126] load generation and benchmarking framework in a third VM instance on Openstack in the trusted zone of the experiment.

Results. We performed 3 experiments using the medical document application introduced in Section 4.6.2. There were in total 8 tactics involved in the benchmarks, namely Mitra, RND, Paillier, and five times DET. Figure 4.8 illustrates *insert* and *equality search* operations, as well as the overall throughput. There is 44% overall throughput loss by employing data protection tactics. Adding our middleware to this setting causes only 1.4% additional overall throughput loss in comparison to the scenario where tactics are inflexibly integrated into the application. The following table further shows the overall average latency, and 50th, 75th and 99th percentile latency. Based on our observation, the execution of aggregate protocols, namely the Paillier PHE, had a considerable impact on these numbers.

Scenario (latency)	50th	75th	99th	Average
S_{A}	62ms	81ms	140ms	85ms
S_{C}	110ms	2500ms	13000ms	828ms
Aggregate	105ms	1700ms	5600ms	1114ms

4.6.4 Cryptographic functions inside database engines

Most protection tactics, especially homomorphic encryption, require some operations to be performed next to the data stored in the cloud [2]. To investigate the feasibility and applicability of running cryptographic functions within the NoSQL database engines in the scope of computing on encrypted data, we have implemented the homomorphic addition of the Paillier cryptosystem as User Defined Functions (UDFs) directly in the underlying database engine. In the current implementation, UDFs are employed for two popular NoSQL databases: Cassandra and MongoDB. The high level homomorphic addition function entails multiplication of the encrypted values m_1 and m_2. For brevity, keys and random values are omitted.

[2]This contribution has been carried out by the author of this dissertation in the scope of the CryptDICE paper [168]

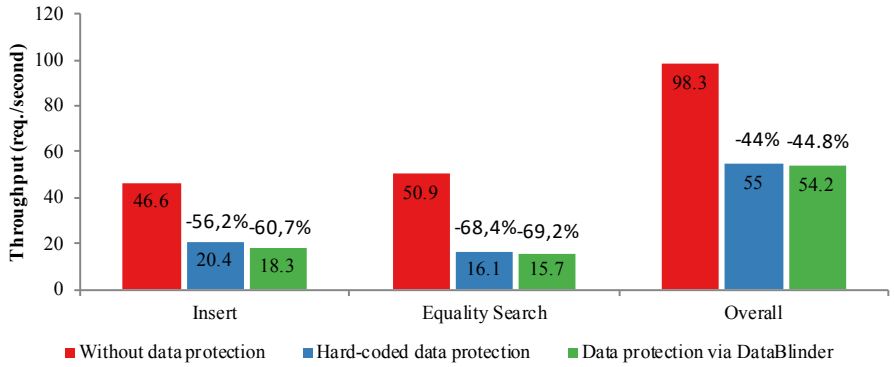


Figure 4.8: Per-operation and overall throughput comparison. Each experiment included ~151k requests, ~50k documents, ~350k secure index operations, and 1,000 benchmark users with a balance between *read* (equality search protocols), *write* (insertions and secure indexing) and *aggregate* operations (search and homomorphic calculation of averages). In addition to the computational and search complexity of each tactic, the Paillier queries were executed ~50k times per run, having a considerable impact on the throughput of the experiments involving data protection tactics (blue and green).

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) \bmod n^2$$

The implementation is based on Javallier [193], a Java library for the Paillier [159] homomorphic scheme. These UDFs can be deployed in MongoDB and Cassandra through both the middleware automatically and the command-line interface manually. In MongoDB, UDFs must be implemented in JavaScript. The multiplication of large integer values requires special types such as `BigInteger` and MongoDB does not support such types for UDFs. Therefore, we employed a custom implementation of this missing type [162]. The function *he_add* is added in the *system* database via a user with privileged access rights. To incorporate these UDFs in queries, we employed the existing Map-Reduce[3] functionality of MongoDB. The `map` function in MongoDB applies to each input document and emits key-value pairs. For keys with multiple values, MongoDB applies the `reduce` function, which collects and condenses the aggregated data. Listing 4.1 illustrates an example query that sums up the amounts of invoices per person in the Billing-as-a-Service SaaS application case.

In Cassandra, we implemented the homomorphic addition as a user-defined aggregate function (UDA). A UDA is typically composed of two functions: a

[3] `https://docs.mongodb.com/manual/core/map-reduce/`

Listing 4.1: An example MongoDB MapReduce query using *he_add* function which implements Paillier addition.

```
db.invoices.mapReduce(
   function() {
      emit(this.name, this.amount);
   }, function(key, values) {
      var total = values[0];
      for (var i = 1; i < values.length; i++){
         total = he_add(total, values[i]);
      }
      return total;
   }
);
```

`state` function to compute the multiplications on each row and update the query state with the results for the next row, and a `final` function to perform some actions at the end such as division for calculation of averages. A typical UDA is applied to data stored in a table as part of query results. To illustrate the idea, the following query goes through all of the invoices in the Billing-as-a-Service use case and calculates the sum of all amounts.

SELECT homsum(*invoice_amount*, *invoice_nsquared*)
FROM *invoices*

Cassandra provides the developers with a wider spectrum of programming languages. We developed the function in Java and leveraged its native `BigInteger` type.

Performance evaluation. The database functions are evaluated in a client-server environment where the client node (running the benchmarking application) interacts with the server node (running the database engine). In our experimental setup, the client node is equipped with Intel Core i7-865U CPU @1.90 GHz @ 2.60 GHz processors with 16 GB RAM and Windows 8 operating system installed. The server node, which is running the database engine is deployed on a private Infrastructure-as-a-service (IaaS) cloud, which is built using OpenStack.

In Cassandra, the computation of a UDF that is performed inside the database engine performs better than the scenario in which the computation is done outside of the database, e.g. inside a VM. For example, to compute homsum on 500K encrypted invoices, the computation inside the database engine took

26,471s and the time to execute in a VM next to the database instance took 53,587s.

Applicability and usability analysis. We achieve a significantly better performance when the computation is moved as close as possible to the data. That means, aggregate queries, which are executed over homomorphically encrypted data from inside the *database engine* are considered to be better prospects for achieving high performance. For this reason, the approach has already been used by other prototyped systems. For example, CryptDB [166], the seminal work in this area, implemented the majority of its functionalities such as adjustable encryption and homomorphic encryption as UDFs in the MySQL database. However, our analysis show that the approach is sub-optimal due to concerns regarding:

Documents	Cassandra	MongoDB
5K	0.441	55.238
10K	0.587	105.994
20K	1.283	255.207
40K	2.058	502.592
80K	3.844	Out of memory error
100K	4.375	Out of memory error
500K	26.471	Out of memory error

Table 4.6: Total execution time in seconds to compute the aggregations, namely homsum in Cassandra and he_add in MongoDB. The mode in which the aggregate function is executed over homomorphically encrypted financial documents from inside the database engine for the MongoDB database leads to the "out-of-memory" problem.

1. *Database-specific performance optimality:* The performance is database-specific as the approach where the computation is performed from inside the *database engine* does not guarantee the best performance for all NoSQL databases. For example, we have also employed UDFs in MongoDB and performed all of the experiments again. The results are presented in Table 4.6, which show that this mode of computation performs worst in most cases. In some cases, it also leads to the *out-of-memory* problem. For instance, to compute he_add on 40K encrypted invoices, this approach takes 502.592 seconds, which is about ~250 times slower than when compared to the results of the Cassandra database. Similarly, we encountered the *out-of-memory* problem when we tried to compute he_add on more than 40k (i.e. from 80K up to 500K) encrypted invoices. The reason lies in the fact that values encrypted by homomorphic encryption

schemes become considerably larger than their plaintext. Therefore, it is crucial how a database system manages the memory and the states in case of an evergrowing homomorphic addition. That varies from database to database depending on the encryption scheme and the architecture of databases.

2. *Limited applicability:* The implementation of UDFs for cryptographic protocols is not always straightforward. Apart from probable security risks (e.g., side-channel attacks), the set of programming primitives offered by programming languages in databases is sometimes limited for our purpose. For example, MongoDB functions should be written in JavaScript. The Paillier homomorphic addition involves multiplication of two relatively large integers, and the size of these integers is dependent on the key size. A safe implementation should use big integers to avoid overflows. At the time of implementing *he_add*, JavaScript did not support such primitives. Therefore, we were obliged to use a custom implementation of arbitrary-length big integers. Furthermore, gaining access to cryptographically secure randomness can also be a challenge. In the MongoDB case, employing external libraries is infeasible, and if external libraries are needed, the source code must be brought to the function implementation.

3. *Increased maintainability of code:* The introduction of external libraries as function implementation within databases may cause unwanted implementation bugs leading to security vulnerabilities. On top of that, it may make the debug and update process cumbersome at scale. Besides, monitoring the database functions at run-time through logging is not a trivial task too. Programming language diversity of database functions causes vendor lock-in and hinders implementation portability and re-usability. Each database comes with its requirements, environment and programming language. Therefore, user-defined functions must be developed and maintained per database. For example, MongoDB supports JavaScript; Redis supports Lua; and Cassandra supports several languages including Java.

In a cloud-native setting where service providers want to employ hosted databases, most NoSQL databases do not offer custom functions due to security and practical reasons. An interesting direction for research would be employing the Function-as-a-Service (FaaS) paradigm although FaaS does not run within databases. However, exploring FaaS-based approaches is not in the scope of our research in this dissertation.

4.7 Conclusion

We presented DataBlinder, a distributed data access middleware that supports fine-grained data protection configuration on application data towards *crypto agility*. Our performance evaluations showed that DataBlinder offers this flexibility at the cost of 1.4% overall throughput loss in comparison to a scenario where the tactics are inflexibly integrated into an application without the middleware.

The current architecture can be deployed as a *cloud-native* service, where the gateway is a stateless data access middleware (e.g., ORM [156]). However, there exist some secure SE tactics, e.g. Sophos [29], requiring keeping the state at the gateway. A challenging research direction towards secure cloud-native systems is to design efficient stateless SE schemes. The current architecture does not take other classes of constructions, e.g., MPC, Oblivious RAM, and TEE, into consideration. It is interesting to explore and abstract their new tradeoffs, different trust models and various execution frameworks.

Chapter 5

Conclusion

This chapter concludes this dissertation. We first summarise the challenges, requirements and contributions in Section 5.1. Section 5.2 discusses the limitations of our contributions and presents the future directions for researchers and practitioners with regards to outsourcing data and computation to the cloud.

5.1 Contributions

This dissertation has addressed several challenges in the scope of outsourcing computation and data to the cloud as outlined in Chapter 1. In this section, we provide a summary of the approaches used to tackle the challenges for each contribution. Table 5.1 presents an overview of the contributions, key challenges, approaches, and evaluations.

Cold start mitigation. The first contribution combined three techniques to mitigate the cold-start latency and reduce the time to start up application containers and pods in Kubernetes. Low cold start latencies are crucial for applications that require elastic scaling, and most importantly when elastic scaling is combined with various SLOs such as the job completion deadline. The layered-based library sharing, i.e. the first technique, encapsulates the external components and software dependencies of an application in various layers of container images; the second technique pre-creates the network infrastructure of application containers within pods to mitigate network setup latency when pods

Contribution	Key challenges	Approach & Evaluation
Cold start mitigation techniques	Latency caused by bootstrapping pods, preparing software environments and user's code affecting applications with deadline-driven SLOs	Extensive experiments using layered-based library sharing, pools of reusable network containers with imperative configuration management
InfraComposer middleware	Engineering workflows with long-running executions & repetitive deployments with complex capacity planning	An adaptive and reflective middleware that uses a step-wise, policy- and history-driven approach forming a MAPE-K loop to optimize the deployment plans; validated extensively with 2 application cases in aeronautics.
DataBlinder middleware	Computing and search on encrypted data with diverse protection tactics; complex and error-prone integration and implementation; crypto agility	A distributed data access middleware with an extensible architecture via the SPI pattern offering encrypted search and partially homomorphic encryption; validated and evaluated with FHIR-compliant medical data; validated the feasibility and applicatbility of using user-defined functions for homomorphic encryption within Cassandra & MongoDB

Table 5.1: Overview of the contributions, key challenges, approaches and evaluation

are supposed to scale-out; and, the third technique employs imperative scaling in Kubernetes, meaning that our scaler communicates directly with the worker nodes and bypasses the Kubernetes controller components. These techniques are explained in Chapter 2.4.

We performed a series of extensive performance evaluations to investigate the effectiveness of these techniques in various combinations. Our findings show that (i) the library sharing approach results in a large reduction in the start-up time of software dependencies, (ii) pre-creating network containers has greater impact when multiple application containers are started in parallel, (iii) the imperative configuration approach introduces start-up time determinism and predictability, making this approach more reliable for applications with SLOs such as job completion deadlines.

InfraComposer. Engineering workflows are composed of various steps, and each step is executed by different software. These workflows aim to simulate and optimise different designs (e.g. MDO), and to achieve this, these workflows are run numerous times iteratively. Execution of these experiments might sometimes take hours or days. Engineers are faced by the following set of non-trivial challenges. First, they must manage a complex cloud deployment required to run the experiments themselves. Second, the cloud deployment requires complex capacity planning to reduce the experiment completion time for each iteration.

This contribution took a step-wise approach. The optimal deployment of the workflows is initially driven by the engineer's annotations, and afterwards based on the execution history of the prior experiments. The middleware and the underlying monitoring system form a MAPE-K loop for optimising the deployment plans. InfraComposer, at the heart of the loop, uses a policy-driven approach to reason about the previous executions. Therefore, engineers are enabled to re-run the workflows and benefit from better deployments tailor-made to the problem at hand.

We implemented and validated the InfraComposer architecture in two industrial use cases. To achieve this, InfraComposer employed the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [195] to compose deployment plans in a modularised and re-configurable way; the deployment and resources meta-models were realised and reified through this mechanism. As validation and evaluation, we presented specific adaptive deployment scenarios in real life in the domain of aeronautics. We validated how both the reflective and the adaptive capabilities of the middleware can cope with each scenario. Our industrial research partner reported [133] that InfraComposer reduced the set-up time. Their workflow performed an extensive design space exploration

analysis, namely the optimisation of the 3D-routing of an automotive cockpit wire harness. Moreover, it improved the scalability and flexibility, and increased the performance of the execution infrastructure by overall ~77.5%.

DataBlinder. Outsourcing sensitive data to the cloud is not a trivial task if the cloud provider is considered to be untrusted. Therefore, if cloud providers do not have access to the encryption keys, data encryption renders the majority of the data operations useless. To protect sensitive data in a pragmatic but secure way, various types of advanced encryption constructions can be used, e.g., searchable encryption for protected search and homomorphic encryption for aggregate operations. Many schemes enable querying on encrypted data with different protection levels and various functionalities. There is a trade-off between security, query expressiveness, and performance. On the one hand, understanding and robust implementation of such cryptographic schemes is a complex task for enterprise software developers; on the other hand, modern distributed software architecture, such as microservices, makes it challenging for security experts to plug in their security policies and mechanisms, especially in the case of advanced cryptographic schemes.

DataBlinder addressed this issue by allowing software developers to annotate the sensitive fields inside the database schemas. These annotations express the required protection level and the query functionalities. To offer a notion of crypto agility in this scope, the middleware, based on the security policies and existing protection tactics, selects and initialises the right tactic for each field. DataBlinder manages these annotations at a per-field granularity. Each field belongs to a collection that can be a part of a separate database. Therefore, the complexity of these cryptographic constructions is abstracted away, and consequently, clients perform data operations as they used to do on plain-text data. To provide an extensible architecture, the DataBlinder tactic selection subsystem is structured using the service provider interface (SPI) pattern. Protection tactics can be extended by implementing the operation interfaces. Based on the commonality between most tactics, a certain set of parameters are injected in the implementations, e.g., key-value database and key management adapters. The middleware executes the tactics' security logic at runtime in a distributed fashion.

Theoretically, moving computation closer to data should result in faster response time. Therefore, we implemented the homomorphic addition of Paillier [159] PHE as user-defined functions (UDF) within the MongoDB and Cassandra databases. In the Cassandra case, our performance evaluations showed that UDF executes in total ~50% faster than the function deployed within a VM next to the database. However, MongoDB, due to its underpinning architecture and the

UDF programming language (JavaScript), could only handle the computation of up to almost 40k documents. Besides, it was considerably slower than Cassandra. We conclude that the database and security community need to rethink the architecture of the existing database systems to support seamless extensions towards privacy-preserving techniques. More concretely, next-generation cloud databases can potentially support various programming runtimes and enable their clients to provide more sophisticated programming logic for their functions. Moreover, the architecture should provide an appropriate source of randomness for the algorithms, decent isolation in terms of performance and security, and flexible dependency management for importing external libraries. Lastly, the execution and distribution of these functions should not be limited to single-node clusters.

To support distributed execution of the tactics in a service-oriented architecture (e.g., microservices), the middleware is prototyped as a gateway and a cloud service. Several searchable encryption schemes with various protection levels and a partially homomorphic encryption scheme have been integrated into the middleware. Our validations and performance evaluations with FHIR-compliant medical data [95] showed that DataBlinder offers this flexibility at the cost of 1.4% overall throughput loss in comparison to a scenario where the tactics are inflexibly integrated into an application without considering the middleware and crypto agility.

Reusability of the approaches. Our contributions are not limited and tailor-made to a single application. This means that we did not leverage generic designs towards specific system requirements. More concretely:

- The techniques used to mitigate the cold start problem are generic and not tied to a very specific software architecture. Employing library sharing, reusable network containers, and imperative configuration management are generic approaches that can be reused for most container-based deployments in the Kubernetes environment. However, we only tested these approaches using event-driven software, in particular job-processing applications. The event-driven software architecture (or deployment model) has been widely used in modern cloud-native applications including those crafted for serverless computing.
- The InfraComposer middleware focuses on engineering workflows. Using the deployment annotations, the generic architecture of the middleware, and the TOSCA [195] deployment specification, the entire cloudification system works for many workflows with similar execution and deployment flows. The middleware reuses pre-built assets and re-configures the

components of existing deployment plans in an adaptive architecture towards efficient executions. We validated this architecture with two real-world engineering workflows from the aeronautics sector.

- The DataBlinder architecture is based on studying two classes of protection tacts, namely searchable encryption, and homomorphic encryption. We outlined the commonality between these tactics and presented generic abstractions accordingly. Moreover, the data-access application programming interface (API), provided to software developers, is based on well-known operations found in most document-oriented and key-value databases. This ensures the validity and reusability of the APIs. We presented a generic abstraction model enabling software developers to annotate database schemas by defining protection level and the required operation. We validated this architecture with a widely-used specification for exchanging healthcare information electronically, known as FHIR [95].

5.2 Limitations and future directions

"*What got you here won't get you there.*"
– Marshall Goldsmith

This dissertation has presented several contributions in the scope of outsourcing computation and data to the cloud. The high-level key research problem of outsourcing such applications to the cloud concerns efficiency and cost-effectiveness in the scope of infrastructure outsourcing, and pragmatic data security in the scope of data outsourcing. This section discusses the current and future directions towards (i) efficient and secure agility of computing instances, where we present an overview of the current and future research directions in the direction of mitigating the cold-start problem, and (ii) cryptographic agility and end-to-end encryption at scale, where we present our views towards the impact of cryptographic agility on software service providers.

5.2.1 Efficient and secure agility of computing instances

Chapter 3 presented a policy-driven middleware which enables engineering workflows to run in the cloud. The key goal was to optimise the deployment plans such that the overall execution becomes more efficient and faster. Our approach relies on the agility of the underlying cloud resources such as virtual machines. Modern software systems move towards decomposing monolithic applications into smaller services, known as micro-services. This decomposition

is driven by various operational and managerial reasons, such as fine-grained scalability. Since OS-level virtualisation is the most common way to package and virtualise services in a lightweight fashion, we therefore in Chapter 2 present three techniques to alleviate the cold-start latency in the scope of the Kubernetes container orchestration framework. With the introduction of serverless computing, in particular Function as a Service (FaaS), the industry and research community have been looking into further decomposing the services into functions. The long-term research idea is to introduce new abstractions for existing and future cloud computing services [161], and ideally move towards a programmable cloud [50, 175]. That means software developers would become less involved in operational matters. To achieve this ideal-world goal, the agility of the cloud resources is of the utmost importance. *The ability to launch and manage these resources as quickly as possible is a key requirement.*

In this subsection, we provide an overview of the ongoing research directions in this domain, and we conclude the discussion with a few remarks on the open security questions.

Moving towards serverless computing. The deployment and execution model of cloud-based software have evolved from traditional virtualisation in the Infrastructure-as-a-Service (IaaS) model to more recent models including serverless computing, also called FaaS. A large number of software service providers, and cloud customers in general, are concerned with the complexity of maintenance, the high degree of resource configurability, and the non-trivial settings for efficient auto-scaling of their software components [177]. Chapter 3 showed that this is not a trivial task. Furthermore, these companies are inclined to focus on their core business rather than managing their low-level deployments. The Berkeley view on serverless computing [107] argues that, in the traditional cloud infrastructure, software developers are most of the time responsible for managing the virtual machines as system administrators. In addition, on the financial side, a subset of their cloud resources often remain under-utilised based on certain workloads, even with the presence of auto-scaling systems. *Therefore, serverless computing is potentially a lucrative execution and deployment model for many businesses.*

Serverless computing is an oxymoron [107] in the sense that cloud functions are still using servers under the hood. However, from the cloud customers' perspective, the computing infrastructure is transparent and delegated to the cloud provider [177]. Therefore, the cloud provider is responsible for the deployments, containers, auto-scaling, and all the other operational tasks.

The cold start *open problem*. Serverless FaaS applications are composed of sequences of cloud functions. In an ideal world, when a function is invoked, clients expect the same response time as the function would have been natively compiled and run on their local workstations. However, in a real-world serverless setting, when a function is invoked, the following steps are executed: a computing node is selected and the relevant container image is fetched; a container is bootstrapped and started; the programming language runtime is launched; the customer function is loaded and initialised; and then the function is ready to handle the user request. Researchers and practitioners have been conducting research to mitigate and minimise the cold start latency to achieve the ideal-world sub-millisecond latency. *This dissertation argues that cold start is an important and unsolved problem in the scope of infrastructure agility, meaning that serverless computing platforms yet strive for a generic solution.*

Research scopes for cold start mitigation and state of the art. We present the state-of-the-art approaches that directly or indirectly alleviate the cold start problem. These research directions are listed in this section since these are still actively being researched and open for innovation. As illustrated in Fig. 5.1, the research scope is summarised into four focus points.

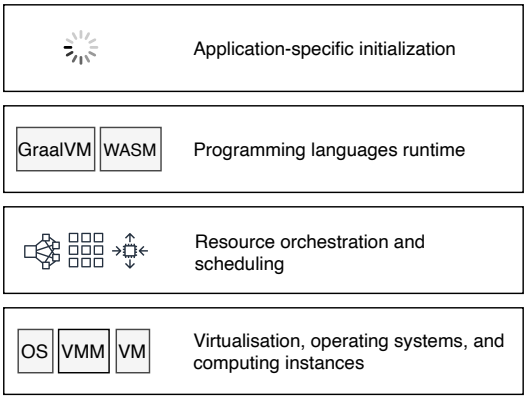


Figure 5.1: The four-layer and full-stack research scope to introduce and optimise computing systems towards mitigating the cold start problem.

1. *Virtualisation, operating systems, and computing instance.* In this layer, the research focus is on improving the low-level computing infrastructure. This layer has been a very active research scope within the systems community, which results in the following areas:

- *Light-weight operating systems.* Starting from Microkernels [79] several decades ago, a branch of Linux kernel design is moving towards reducing the kernel modules to a bare minimum, i.e. a minimal stack of libraries enough to bootstrap the OS and run the applications. The ongoing research efforts include Unikernel [130], MicroVMs [3], and more. For example, Unikernel allows developers to compose and compile a minimal set of libraries and their applications to create sealed and fixed-purpose images. These images can run directly on a hypervisor. Next to their potential benefits such as improved security, small footprints, maximizing hardware utilisation, this line of Linux kernels is relatively faster to boot up.
- *Optimising containers.* This approach aims to optimise the kernel and container runtimes towards faster containers. Oakes et al. [151] present a task provisioning system that includes the optimisation of containers. They replace the flexible and costly mechanisms with simpler alternatives in the scope of disk, network, and user namespaces. Moreover, they provision new runtimes by forking from the Zygote processes instead of creating new ones from scratch. That would eliminate the interpreter and package initialisation cost.

2. *Resource orchestration and scheduling.* In this layer, the orchestration framework, including the schedulers and auto-scalers, are the focus point for further research to improve the agility of cloud functions. The three categories of approaches include:

- *Resource pooling.* This approach aims to reuse computing resources to improve the cold-start latency. There exist various pools, including: (i) the pool of warm containers in which containers are ready to serve the clients' requests, (ii) the pool of containers where pre-warmed containers are started but the application is not yet initialised, and (iii) the pool of network containers in which the network infrastructure of the application containers is pre-created. The plain usage of these techniques results in a large memory footprint. Therefore, the research challenge is to find a reasonable and cost-effective tradeoff between sub-second startup time versus resource underutilisation. The notion of pool or queue is presented in several works [180, 145, 137, 123, 182, 18, 179]. For example, to reduce the excessive resource consumption of the unused resources in the pools, some work [179, 182] proactively spawn containers in the pool based on the execution flow of the cloud functions or the workload pattern.
- *Snapshotting and caching.* This line of approaches aims to present cache-aware schedulers to start cloud functions on the compute nodes in which relevant resources are already available. The cached resources include container images or even snapshots of a cloud function. A recent work by Silva et al. [180] employ the checkpoint-recovery technique to create

snapshots of containers at certain points in time. Cloud function snapshots, e.g. made by the tool CRIU [124], are Linux processes snapshots since containers are just processes from the host operating systems perspective. That means restoring a snapshot is considerably fast; therefore, the outcome leads to the improved startup time of the functions. Section 2.3 presents further related research in this scope.

- *Execution-flow based.* In FaaS, applications are composed of collections of cloud functions. The control flow of the applications typically includes various subsets of these functions for different aspects of the business logic. Therefore, at runtime, we have various execution flows. Several work [182, 57] present new techniques to provision the cloud functions involved in the execution flows to alleviate the cold start problem. For example, Daw et al. [57] target the cascading cold-start problem in an execution path. They present an approach to detect the most likely execution path and speculatively deploy the containers.

3. *Programming languages runtime.* When a container is launched, to execute an application, the language runtime is required to be launched and initialised. Recent research shows that some popular language runtimes such as Java, Python, or JavaScript can be very slow [151, 18] to bootstrap. For instance, Java Virtual Machine (JVM) enables Java to be platform-independent by compiling byte-codes to machine code at runtime; this process is known as just-in-time (JIT) compilation. Although this approach presents adaptively further optimisations at runtime, the startup time is slow. Ahead-of-time (AOT) compilation would eliminate the JVM layer consuming around 300 MB of memory [142]. Memory is very costly in the current commercial FaaS offerings [142]. Therefore, by using AOT compilation, higher-level programming languages such as Java can be compiled to native machine code. As a result, the startup time of such applications can become considerably faster. One of the major efforts in this scope is GraalVM [158].

4. *Application-specific initialisation.* Optimising the infrastructure stack towards faster startup time mostly results in constant time improvements. This includes improving the hardware, operating systems, virtualisation, orchestrators, and execution runtimes. However, the bootstrapping time of cloud functions is shown to be application specific [179]. In the initialisation phase, functions can be blocked by calling different resources such as other functions, and databases. An application might require a time-consuming machine-learning model initialisation at startup time. We highlight, among others, two research directions in this domain:

- *Library sharing.* Oftentimes, depending on the workload, it is required to spawn several instances of an application container. As result, a large

portion of the software dependencies such as libraries is redundantly loaded in memory. Consequently, this potentially causes high and unnecessary memory consumption as well as a slower startup time. The research goal is to share software dependencies between identical containers. Recent work [73, 18] improves the cold-start problem by taking this goal into consideration.

- *Workload based.* Recent research published by Microsoft Research [179] reports that application workloads are considerably diverse due to the heterogeneity of applications and varying request arrivals; their observations are based on the entire production FaaS workload of Azure Functions. For instance, popular applications consume more memory; therefore, having warm pools of functions at scale is not always an option. Azure has a 20min keep-alive time policy for the functions. A function with a periodic request arrival of every 21min would suffer from the cold-start latency upon each request. Interestingly, more than 75% of functions execute up to a maximum of 10 seconds, which is roughly equal to the cold-start duration on average. The authors create histograms based on the frequency of the invocations per application, and they pre-warm the resources just in time as well as adjusting the keep-alive time adaptively. For the workloads that cannot be captured by the histograms, their system employs time-series analysis to proactively pre-warm the containers. This cold-start mitigation approach is realistic and promising for further research; however, it requires the workloads to be publicly available to the research community. Fortunately, Microsoft has made a sanitised portion of the workload traces available to the public [170].

Further research on composable and hybrid techniques. There are many approaches to mitigate and alleviate the cold-start latency. Although each of these approaches is a subject of ongoing research, it is shown that applications themselves, and their workload [170] are of the utmost importance. The research community should further investigate the possibility of *hybrid* techniques. For example, by looking into the workload traces of a specific application, a FaaS provider can speculatively spawn warm pools of resources. This line of research requires the industry to collaborate with academia. For instance, Microsoft has published a portion of Azure Functions workload traces to the public [170]. In the long term, it would be interesting to look into advanced middleware frameworks to infer the properties and characteristics of FaaS applications and *compose* the right approaches for the task at hand.

Concluding remarks on security. As presented above, the community has been conducting research on the cold-start problem at all layers. However, the security

threats are not always considered and are classically left as an afterthought. For example, the snapshot family of approaches uses the checkpoint-recovery technique to make a snapshot of a warm container process, and restore it at the time of need. If this is not done correctly, it might become a recipe for security complications [139]. For instance, in the snapshotting approach, the following questions are crucial to be answered:

- Snapshots typically preserve the memory address structure. If the memory address space layout randomisation (ASLR) is intended to be used, What is the strategy if we restore containers from the snapshots?
- Based on the security best practices, containers should hold no application secrets; however, how do we guarantee that the snapshots, taken at a certain point in time, contain no secret key in memory?
- Linux random number generators rely on entropy pools. Do snapshots bring along the same entropy pools?

The Linux-based OS-level virtualisation, e.g. Docker containers, are considered to be about abstraction rather than the traditional isolation that we are familiar with in the scope of virtual machines. Most Linux containers use `cgroups` to limit the memory and CPU usage, and kernel `namespaces` to isolate the operating environment such as process trees, network, and more. However, the OS-level virtualisation relies on a shared kernel. Therefore, security assurance is typically based on the security ecosystem but not on isolation. The cloud practitioners, e.g. Netflix [199], typically achieve security through cloud-based offerings and application-level security techniques, such as IAM, access control sidecars, network security groups, securing the container orchestration frameworks, mutual TLS between nodes, and more.

Hardening the host machines and the containers is of the utmost importance because a kernel vulnerability might put everything at risk (e.g. CVE-2016-5195 [150]). In that regard, recent research towards light-weight operating systems such as Unikernels [130], MicroVMs [3], Kata Containers [55], Nabla Containers [203], and more, presents new approaches to either reduce the attack surface by eliminating the unnecessary kernel modules or providing VM-grade isolation by running dedicated kernels. This is an ongoing research direction to find lightweight and at the same time secure alternatives to the classic VMs and containers.

Next to the aforementioned research challenges, large-scale service providers, e.g. Netflix [199], also including cloud providers, require vulnerability management processes. The major task is to continuously scan the assets and services for the known vulnerabilities, and consequently take an action against the stale assets.

In our scope, assets can be container instances, images, snapshots, caches, and more. These assets might either be present in the asset registries, in the warm pools or even cached in the nodes. Therefore, further applied research is required to present new mechanisms to securely flush out the vulnerable components in the development and production settings. This might seem trivial; however, a large-scale service provider with thousands of assets would eventually suffer if we optimise our infrastructure towards low-latency startups without paying attention to security. For instance, if a complex library-sharing approach is in place, and hundreds of production containers are in operation using this system, what would be the most secure and cost-effective approach to gracefully upgrade a vulnerable library?

The community should take a step back and revisit the existing techniques with regards to security implications, as well as including security analysis as a requirement for future techniques.

5.2.2 Cryptographic agility and end-to-end encryption at scale

In Chapter 4, the DataBlinder architecture aimed to bring software developers and security experts closer to each other by introducing policy-driven abstractions for data protection techniques. In this subsection, we present how software systems that lack cryptographically agile software architecture are affected by necessary cryptographic upgrades. In particular, we discuss how applications are affected, and the potential impact on SLA-driven applications. We further present the importance of observability and discoverability of cryptographic assets as well as cryptographic agility for confidential computing. Moreover, we discuss various views regarding cryptographic agility, and we present our views on the subject, especially regarding DataBlinder. We conclude this section by presenting the requirement for the protection of primary keys and data replication for highly scalable databases.

The potential impact of cryptographic agility on software systems. RFC-7696 [98] defines cryptographic agility as the easy migration from an algorithm suite to another one over time. Such a migration is often motivated by advances in crypto-analytic attacks and computing capabilities. The migration is not as simple and smooth as it might seem. For instance, it has been shown that any change in the following parameters have caused the companies and the software communities tremendous efforts:

- *Different output lengths.* Oftentimes the output length of cryptographic functions changes depending on the underpinning constructions. For

example, the MD5 hash function outputs 128 bits and it is completely broken today [198]. The SHA-2 family of hash functions, as a replacement, has the output size of 224 and 256 bits in the shortest form [89]. The way the output is processed in the entire software stack, and the way it is persisted in memory and databases are subject to cascading difficulties across all layers. Therefore, such software systems are considerably hard to be refactored.

- *Different key lengths.* Depending on the type of cipher suite as well as the security strength, the key length varies and potentially causes engineering and operational challenges. For example, software systems tailored with 512-bit or even 1024-bit RSA are not considered secure anymore. Today, the National Institute of Standards and Technology (NIST) recommends [12] 2048-bit RSA keys that have a security equivalence to 112-bit symmetric keys, and 15360-bit length equal to 256-bit symmetric keys. Upgrading key size in many software systems and networked assets is not a trivial task because of many reasons such as the lack of configurability.
- *New input parameters.* The type and number of input parameters are subject to change. For example, the adoption of elliptic-curve cryptography has changed many aspects of the cryptographic primitives including input parameters due to its relatively different underlying mathematical context. In a rigid architecture, such a change affects some parts of the software systems, or even other external software artifacts consuming the interfaces. Therefore, the change impact, also known as ripple effect in software engineering, might potentially get beyond an application context.

The notion of cryptographic performance and service layer agreements (SLA). A cryptographic interface can change in several dimensions: function names, input-output lengths, input-output parameters, and naturally the computational complexity. Software service providers often agree upon certain service-level objectives (SLO) with their customers (e.g. the service throughput). Therefore, changing a cryptographic construction by using a different underpinning mathematical building block or even a larger key length often affects the entire stack. For instance, the performance of the public-key cipher-suites varies depending on the mathematical hardness assumption such as the integer factorisation, discrete logarithm, or the lattice-based problems. At the higher layers of the software stack, this often can potentially be observed as extra latency and degraded throughput. Conversely, the cryptographic primitives might become more efficient and lightweight; an example can be the transition from the RSA family of schemes to ecliptic-curve crypto-systems. As a result, the computing resources might become under-utilised and over-allocated for the task at hand. This issue might seem trivial and unimportant; however,

capacity planning and resource allocation are financially crucial for large-scale service providers. Consequently, cryptographic agility requires further applied research towards smarter performance engineering by optimising the existing hardware to maximise the CPU cycles, concurrency, and lastly the autoscaling systems.

Observability and discoverability of cryptographic assets. Cryptographic assets such as key pairs and certificates, if not managed in a centralised fashion, might potentially cause difficulties in the process of upgrading the protocols. For instance, recently ABN AMRO Bank reported [191] that it took the organisation almost more than a year to discover and replace their obsolete certificates. The bank produces thousands of certificates for their customers, employees, internal and external software. This process was motivated by the deprecation of `SHA-1`. This problem can also potentially occur in large-scale software systems built using microservice architecture. There is an open direction for applied research in the scope of observability of security assets across thousands of services. It would be interesting to know how those assets can be identified, tracked, and upgraded at the time of need, and that should happen in a graceful manner without service outage.

Cryptographic agility for confidential computing. Cryptographic changes in software-only solutions are intuitively manageable. With the presence of flexible software architecture, changes can be done more swiftly in comparison to the hardware-software security solutions. In the modern and new generation of hardware architecture, various hardware-based components assist operating systems, hypervisors, and software systems to be secure in untrusted environments. Examples can be secure cryptoprocessors such as trusted platform modules (TPM), hardware security modules (HSM), physical unclonable functions (PUF), trusted execution environments (TEE), or even the `AES-NI` instruction set. Major corporations have already integrated these hardware-software technologies in their offerings, such as the extensive usage of TPM in the IBM Power9 processor chips to support secure VMs [99], or Microsoft Azure that uses TEEs to offer confidential computing [172] to their cloud customers. The cryptographic agility landscape of the next-generation software, built on top of these technologies, is not clear. More applied research is required to understand the impact and the strategic action points in response to vulnerabilities or critical incidents. This is important since the semiconductor industry typically has a slower pace in patching or replacing the vulnerable components in comparison to the software industry.

Cryptographic agility, the divided community, and our views. A part of the security community does not consider cryptographic agility to be a right and safe approach to design cryptographic protocols. The *first* argument stems from the idea that cryptographic primitives are not the root cause of the security vulnerabilities in real-world systems; in fact, the way these primitives are combined to form larger protocols are often the reason for the security breaches [9]. The *second* argument is about the fact that software developers often make mistakes in the implementation of the protocols due to the complexity of cryptographic configurations. And, if architecture is flexible to allow software developers to switch cipher suites themselves, it is considered to be rather harmful, especially the types of crypto agility solutions that work at runtime through protocol negotiations. As an example, in the past, an attacker could negotiate the use of a lower version of TLS in OpenSSL allowing attackers to mount downgrade attacks [7]. Therefore, they recommend that cryptographic algorithms should be baked in the software modules along with version numbers. Therefore, the algorithms should be swapped under the hood with new version numbers. Our views with regards to these arguments are as follows:

- It is correct that the combination of cryptographic primitives can potentially leak unwanted information; however, it is incorrect to neglect the security of the primitives themselves. There have been several points in time that the industry had to upgrade the existing schemes to secure alternatives, such as `DES`, `3DES`, `MD5`, `SHA1`, and so forth. For instance, many companies including banks [191] have already started looking into their software and infrastructure architecture for post-quantum schemes.
- These arguments primarily focus on runtime negotiations and protocol updates; however, the actual focus of cryptographic agility is beyond the runtime state. In fact, a large part of such upgrades happens at the deployment time since software projects often require considerable changes. For the algorithm-transition mechanisms based on negotiation-based approaches, their concern is valid [9]; therefore, extra care is required to be taken into consideration because of the risks of downgrade attacks.
- Research results [117] present that 83% of the bugs are in applications that misuse cryptographic libraries. However, this does not justify that cryptographic schemes, artifacts, and configurations should be tailored to the architecture of an application in a rigid manner. Having a correct level of abstraction is crucial. Libraries such as Libsodium [121] or NaCl [19] aim at offering cryptographic functions to the developers with usability and simplicity in mind. Yet, the advances in program verification open up new research avenues to eventually meet stronger security guarantees outside of the scope of the cryptographic libraries too.

Cryptographic agility, DataBlinder, and open directions. DataBlinder, presented in Chapter 4, aims to bring software developers and security experts closer, and yet abstracting away the cryptographic complexities. We present an abstraction model for data operations towards usability and simplicity of defining required protection levels. Developers only require to define their required operation and protection level, and the middleware handles the right tactic selection at runtime. *Tactic selection at runtime* might imply the agility of the cryptographic constructions; however, this aspect does not entail tactic upgrade at runtime. Tactic upgrade at runtime means replacing a tactic that is already serving the clients to a new version or a completely new tactic. This is one of the key requirements of cryptographic agility, and it is potentially an open direction for further systems research.

Although DataBlinder, in the current version, does not consider migration, it lays out the software architecture in a way in line with cryptographic agility of such systems: (i) the abstraction models simplify the API's; (ii) the extensible architecture enables future tactics to be integrated with less impact on the middleware codebase; and (iii) by using the *gateway* pattern and the data-access abstractions, the current deployment model decouples the cryptographic constructions from the rest of the microservice deployment. As a result, the latter reduces the ripple effect in case of any change in the algorithms. Lastly, *cryptographic configurability* of software is an important aspect of crypto agility [98, 47]. In this regard, DataBlinder enables security experts to control the tactic selection mechanisms through a policy-driven approach.

Breach-resistant primary key and replication protection for highly scalable databases. Chapter 4 and the CryptDICE [168] middleware present a solution to allow developers to seamlessly outsource sensitive data to the cloud in a protected way; however, the protection of primary keys is not fully investigated. In SQL and NoSQL databases, primary keys are used for the identification and the lookup process of the documents. More importantly, these values are the basis for correlations between different collections of documents, known as foreign keys in the relational database systems. Modern highly scalable database systems leverage primary keys and employ partitioning techniques such as *consistent hashing* [190, 206] to distribute data evenly across a large-scale cluster of database nodes. Moreover, the partitioning algorithms facilitate the process of data replication across the nodes and data centers. The recent Facebook data breach [96] has shown the importance of protecting such data against passive, or snapshot adversaries. The current state of the art have looked into protecting foreign keys [166, 85, 88, 143, 178, 111] and distributed hash tables (DHT) [4, 207]. However, we have not yet seen any industry adoption and extensive research by the systems community to understand how real-world

systems are affected by these techniques. Furthermore, not much attention has been paid to data replication based on primary keys. Therefore, to protect data in highly scalable systems, further research should be done towards the next generation of secure partitioning and replication strategies.

Bibliography

[1] Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 1–35.

[2] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., and Stransky, C. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 154–171.

[3] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.

[4] Agarwal, A., and Kamara, S. Encrypted distributed hash tables. *IACR Cryptol. ePrint Arch. 2019* (2019), 1126.

[5] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, pp. 563–574.

[6] Al-Dhuraibi, e. a. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing* (2017).

[7] Alashwali, E. S., and Rasmussen, K. What's in a downgrade? a taxonomy of downgrade attacks in the tls protocol and application protocols using tls. In *International Conference on Security and Privacy in Communication Systems* (2018), Springer, pp. 468–487.

[8] Amann, S., Nadi, S., Nguyen, H. A., Nguyen, T. N., and Mezini, M. Mubench: A benchmark for api-misuse detectors. In *Proceedings of*

the 13th International Conference on Mining Software Repositories (2016), pp. 464–467.

[9] Arciszewski, S. Against cipher agility in cryptography protocols, 2019. `https://paragonie.com/blog/2019/10/against-agility-in-cryptography-protocols` (Last visit: 2021-06-17).

[10] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[11] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, 2017, pp. 1–20.

[12] Barker, E., and Dang, Q. NIST special publication 800-57 part 1, revision 4. *NIST, Tech. Rep 16* (2016).

[13] Bellare, M., Boldyreva, A., and O'Neill, A. Deterministic and efficiently searchable encryption. Cryptology ePrint Archive, Report 2006/186, 2006. `https://eprint.iacr.org/2006/186`.

[14] Beni, E. H., Lagaisse, B., and Joosen, W. WF-interop: adaptive and reflective rest interfaces for interoperability between workflow engines. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware* (2015), ACM, p. 1.

[15] Beni, E. H., Lagaisse, B., and Joosen, W. Adaptive and reflective middleware for the cloudification of simulation & optimization workflows. In *Proceedings of the 16th Workshop on Adaptive and Reflective Middleware* (2017), pp. 1–6.

[16] Beni, E. H., Lagaisse, B., and Joosen, W. Infracomposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows. *Journal of Systems Architecture 95* (2019), 36–46.

[17] Beni, E. H., Lagaisse, B., Joosen, W., Aly, A., and Brackx, M. Datablinder: A distributed data protection middleware supporting search and computation on encrypted data. In *Proceedings of the 20th International Middleware Conference Industrial Track* (2019), pp. 50–57.

[18] Beni, E. H., Truyen, E., Lagaisse, B., Joosen, W., and Dieltjens, J. Reducing cold starts during elastic scaling of containers in kubernetes.

In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021), pp. 60–68.

[19] Bernstein, D. Nacl, 2008. `https://nacl.cr.yp.to/` (Last visit: 2021-06-17).

[20] Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[21] Biryukov, A., Velichkov, V., and Corre, Y. L. Automatic search for the best trails in arx: Application to block cipher Speck. Cryptology ePrint Archive, Report 2016/409, 2016. `https://eprint.iacr.org/2016/409`.

[22] Blair, G., Costa, F., Coulson, G., Delpiano, F., Duran, H., Dumant, B., Horn, F., Parlavantzas, N., and Stefani, J.-B. The design of a resource-aware reflective middleware architecture. In *Meta-Level Architectures and Reflection* (1999), Springer, pp. 115–134.

[23] Blair, G. S., Coulson, G., Robin, P., and Papathomas, M. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (2009), Springer-Verlag, pp. 191–206.

[24] Blochberger, M., Petersen, T., and Federrath, H. Mitigating cryptographic mistakes by design. *Mensch und Computer 2019-Workshopband* (2019).

[25] Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. Order-preserving symmetric encryption. Cryptology ePrint Archive, Report 2012/624, 2012. `https://eprint.iacr.org/2012/624`.

[26] Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., and Zimmerman, J. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation, 2015.

[27] Bösch, C., Hartel, P., Jonker, W., and Peter, A. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR) 47*, 2 (2014), 1–51.

[28] Bösch, C., Hartel, P., Jonker, W., and Peter, A. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR) 47*, 2 (2015), 18.

[29] Bost, R. Sophos: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1143–1154.

[30] Bost, R., Minaud, B., and Ohrimenko, O. Forward and backward private searchable encryption from constrained cryptographic primitives. Cryptology ePrint Archive, Report 2017/805, 2017. `https://eprint.iacr.org/2017/805`.

[31] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT) 6*, 3 (2014), 1–36.

[32] Breitenb ucher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., and Wettinger, J. Combining declarative and imperative cloud application provisioning based on TOSCA. In *2014 IEEE International Conference on Cloud Engineering* (2014), pp. 87–96.

[33] Brewer, E. A. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing* (2015), pp. 167–167.

[34] Browne, P. *JBoss Drools business rules*. Packt Publishing Ltd, 2009.

[35] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, p. 991–1008.

[36] Burns, J., Moore, D., Ray, K., Speers, R., and Vohaska, B. Ec-oprf: Oblivious pseudorandom functions using elliptic curves. *IACR Cryptology ePrint Archive 2017* (2017), 111.

[37] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems 25*, 6 (2009), 599–616.

[38] Byrro, R. Can we solve serverless cold starts?, 2019.

[39] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. Seuss: Rapid serverless deployment using environment snapshots. *arXiv preprint arXiv:1910.01558* (2019).

[40] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. Seuss: Rapid serverless deployment using environment snapshots. *arXiv preprint arXiv:1910.01558* (2019).

[41] Caíno-Lores, S., Lapin, A., Kropf, P., and Carretero, J. Methodological approach to data-centric cloudification of scientific iterative workflows. In *International Conference on Algorithms and Architectures for Parallel Processing* (2016), Springer, pp. 469–482.

[42] Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), ACM, pp. 668–679.

[43] Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.-C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. Cryptology ePrint Archive, Report 2014/853, 2014. https://eprint.iacr.org/2014/853.

[44] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., and Steiner, M. Highly-scalable searchable symmetric encryption with support for boolean queries. Cryptology ePrint Archive, Report 2013/169, 2013. https://eprint.iacr.org/2013/169.

[45] Cash, D., and Tessaro, S. The locality of searchable symmetric encryption. Cryptology ePrint Archive, Report 2014/308, 2014. https://eprint.iacr.org/2014/308.

[46] Chamani, J. G., Papadopoulos, D., Papamanthou, C., and Jalili, R. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1038–1055.

[47] Chen, L., LaMacchia, B., Ott, D., and Ramzan, Z. Rsa conference panel: Cryptographic agility: Anticipating, preparing for and executing change, 2020. https://www.youtube.com/watch?v=8pGJVTekDyM (Last visit: 2021-06-17).

[48] Chen, T., and Bahsoon, R. Self-adaptive and online qos modeling for cloud-based software services. *IEEE Transactions on Software Engineering 43*, 5 (2017), 453–475.

[49] Chen, T., Bahsoon, R., and Yao, X. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR) 51*, 3 (2018), 61.

[50] Cheung, A., Crooks, N., Hellerstein, J. M., and Milano, M. New directions in cloud programming. *arXiv preprint arXiv:2101.01159* (2021).

[51] Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology* (2018), 1–58.

[52] Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology 33*, 1 (2020), 34–91.

[53] CipherCloud. Ciphercloud. Online, 2014. `https://www.ciphercloud.com/`.

[54] Civolani, L., Pierre, G., and Bellavista, P. Fogdocker: Start container now, fetch image later. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing* (2019), pp. 51–59.

[55] Containers, K. Kata containers, 2021. `https://katacontainers.io` (Last visit: 2021-06-17).

[56] Curtmola, R., Garay, J., Kamara, S., and Ostrovsky, R. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security 19*, 5 (2011), 895–934.

[57] Daw, N., Bellur, U., and Kulkarni, P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 356–370.

[58] DB-Engines. Db-engines ranking of relational dbms, 2021.

[59] Demertzis, I., and Papamanthou, C. Fast searchable encryption with tunable locality. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1053–1067.

[60] Derek, P., and Kapil, T. Understanding TOSCA and containers. *OASIS* (2014).

[61] Diallo, M. H., Hore, B., Chang, E.-C., Mehrotra, S., and Venkatasubramanian, N. Cloudprotect: managing data privacy in cloud applications. In *2012 IEEE Fifth International Conference on Cloud Computing* (2012), IEEE, pp. 303–310.

[62] Dutreilh, X., Moreau, A., Malenfant, J., Rivierre, N., and Truck, I. From data center resource allocation to control theory and back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (2010), IEEE.

[63] El-Rewini, H., and Abd-El-Barr, M. *Advanced computer architecture and parallel processing*, vol. 42. John Wiley & Sons, 2005.

[64] ElGamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory 31*, 4 (1985), 469–472.

[65] European Commission. Regulation eu 2016/67.

[66] Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., and Steiner, M. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Report 2015/927, 2015. `https://eprint.iacr.org/2015/927`.

[67] Ferreira, J. B., Cello, M., and Iglesias, J. O. More sharing, more benefits? a study of library sharing in container-based infrastructures. In *European Conference on Parallel Processing* (2017), Springer, pp. 358–371.

[68] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. Stack overflow considered harmful? the impact of copy paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 121–136.

[69] Fu, S., Mittal, R., Zhang, L., and Ratnasamy, S. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)* (2020).

[70] Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., and Sadeghi, A.-R. Hardidx: Practical and secure index with sgx. In *IFIP Annual Conference on Data and Applications Security and Privacy* (2017), Springer, pp. 386–408.

[71] Fuller, B., Varia, M., Yerukhimovich, A., Shen, E., Hamlin, A., Gadepally, V., Shay, R., Mitchell, J. D., and Cunningham, R. K. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 172–191.

[72] Furtado, T., Francesquini, E., Lago, N., and Kon, F. A middleware for reflective web service choreographies on the cloud. In *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware* (2014), ACM, p. 9.

[73] Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L., and Parmer, G. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 265–279.

[74] Gamma, E. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[75] Gigaspaces. Cloudify, 2018.

[76] Godard, S. iostat(1) - linux man page.

[77] Goh, E.-J., et al. Secure indexes. *IACR Cryptology ePrint Archive 2003* (2003), 216.

[78] Goldreich, O. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), pp. 182–194.

[79] Golub, D. B., Julin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G., and Bohman, D. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (1992), pp. 11–30.

[80] Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. *Data structures and algorithms in Java*. John Wiley & Sons, 2014.

[81] Grace, P., Coulson, G., Blair, G. S., and Porter, B. A distributed architecture meta-model for self-managed middleware. In *Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM'06)* (2006), ACM.

[82] Gregg, B. Thinking methodically about performance. *Communications of the ACM 56*, 2 (2013), 45–51.

[83] Grubbs, P., Lacharité, M.-S., Minaud, B., and Paterson, K. G. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE Symposium on Security and Privacy (S&P) 2019* (2019).

[84] Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., and Ristenpart, T. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 655–672.

[85] Hahn, F., Loza, N., and Kerschbaum, F. Joins over encrypted data with fine granular security. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), IEEE, pp. 674–685.

[86] Han, R., Ghanem, M. M., Guo, L., Guo, Y., and Osmond, M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems 32* (2014), 82–98.

[87] Han, R., Guo, L., Ghanem, M. M., and Guo, Y. Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* (2012), IEEE.

[88] Hang, I., Kerschbaum, F., and Damiani, E. Enki: access control for encrypted query processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 183–196.

[89] Hansen, T., and 3rd, D. E. E. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[90] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 181–195.

[91] Hasan, M. Z., Magana, E., Clemm, A., Tucker, L., and Gudreddi, S. L. D. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE* (2012), IEEE.

[92] Herbst, N., Krebs, R., Oikonomou, G., Kousiouris, G., Evangelinou, A., Iosup, A., and Kounev, S. Ready for rain? a view from spec research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470* (2016).

[93] Herbst, N. R., Kounev, S., and Reussner, R. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing ({ICAC} 13)* (2013), pp. 23–27.

[94] Heydari Beni, E., Dieltjens, J., Truyen, E., Lagaisse, B., and Joosen, W. Reducing cold starts during elastic scaling of containers in kubernetes. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021), ACM.

[95] HL7 health-care standards organization. Fast healthcare interoperability resources, 2011. `http://hl7.org/fhir/` (Last visit: 2019-05-16).

[96] Holmes, A. 533 million facebook users' phone numbers and personal data have been leaked online, 2021. `https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4?r=US&IR=T` (Last visit: 2021-06-17).

[97] Hoogreef, M. Advise, formalize and integrate mdo architectures: A methodology and implementation.

[98] Housley, R. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. RFC 7696, Nov. 2015.

[99] Hunt, G. D., Pai, R., Le, M. V., Jamjoom, H., Bhattiprolu, S., Boivie, R., Dufour, L., Frey, B., Kapur, M., Goldman, K. A., et al. Confidential computing for openpower. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 294–310.

[100] imec.icon. Seclosed: Secure, cloud-based storage and processing of sensitive documents, 2016-2018. `https://www.imec-int.com/en/what-we-offer/research-portfolio/seclosed` (Last visit: 2021-03-30).

[101] Ionic. Introducing ionic encrypted search. Online, 2014. `https://www.ionic.com/blog/introducing-ionic-encrypted-search/`.

[102] Ishai, Y., Kushilevitz, E., Lu, S., and Ostrovsky, R. Private large-scale databases with distributed searchable symmetric encryption. In *Topics in Cryptology - CT-RSA 2016* (Cham, 2016), K. Sako, Ed., Springer International Publishing, pp. 90–107.

[103] Information technology - Syntactic metalanguage - Extended BNF. Standard, International Organization for Standardization and International Electrotechnical Commission, Geneva, CH, Dec. 1996.

[104] ITEA3. Idealism: Integrated and distributed engineering services framework for mdo, 2014-2018. `https://itea3.org/project/idealism.html` (Last visit: 2021-03-20).

[105] Jensen, S. K., Pedersen, T. B., and Thomsen, C. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering 29*, 11 (2017), 2581–2600.

[106] Jiang, Y., Perng, C.-s., Li, T., and Chang, R. Self-adaptive cloud capacity planning. In *2012 IEEE Ninth International Conference on Services Computing* (2012), IEEE, pp. 73–80.

[107] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[108] Kacsuk, P., Kovács, J., and Farkas, Z. The flowbster cloud-oriented workflow system to process large scientific data sets. *Journal of Grid Computing* (2018).

[109] Kamara, S., and Moataz, T. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2017), Springer, pp. 94–124.

[110] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (2003), 41–50.

[111] Khazaei, S., and Rafiee, M. Towards more secure constructions of adjustable join schemes. *IEEE Transactions on Dependable and Secure Computing* (2020).

[112] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1–19.

[113] Kubernetes. Kubernetes documentation, 2020.

[114] Kulkarni, A. R. Development of knowledge based engineering tool to support finrudder interface design and optimization. *Master's thesis, Delft University of Technology, Faculty of Aerospace Engineering* (2015).

[115] Lab, E. S. Clusion, 2016. `https://github.com/encryptedsystems/Clusion` (Last visit: 2019-04-30).

[116] Lacharité, M.-S., Minaud, B., and Paterson, K. G. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 297–314.

[117] Lazar, D., Chen, H., Wang, X., and Zeldovich, N. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014), pp. 1–7.

[118] Lee, F. Architectural points of extension and scalability for the elk stack. `https://fabianlee.org/2016/11/28/`. Accessed: 2018-08-27.

[119] Lewi, et al. An implementation of order-revealing encryption, 2016. `https://github.com/kevinlewi/fastore` (Last visit: 2019-05-16).

[120] Lewis, I. The almighty pause container, 2017.

[121] Libsodium. Implementing a crypto services strategy at abn amro bank, 2015. `https://doc.libsodium.org/` (Last visit: 2021-06-17).

[122] Lin, P.-M., and Glikson, A. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221* (2019).

[123] Lin, P.-M., and Glikson, A. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221* (2019).

[124] Linux Software. Criu: checkpoint or restore functionality for linux.

[125] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 973–990.

[126] Locust. Locust: An open source load testing tool. `https://locust.io/`. [Last visited on September 28, 2020].

[127] Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing 12*, 4 (2014), 559–592.

[128] M. R. Alves, P. G., and Aranha, D. F. A framework for searching encrypted databases. *Journal of Internet Services and Applications 9*, 1 (Jan 2018), 1.

[129] Macías-Escrivá, F. D., Haber, R., Del Toro, R., and Hernandez, V. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications 40*, 18 (2013), 7267–7279.

[130] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News 41*, 1 (2013), 461–472.

[131] Madkour, A. Order-preserving encryption, 2018. `https://github.com/aymanmadkour/ope` (Last visit: 2019-05-16).

[132] Manner, J., Endress, M., Heckel, T., and Wirtz, G. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (2018), IEEE, pp. 181–188.

[133] Marco Panzeri, Emad Heydari Beni, B. L. R. D. M. M. F. S. Smart deployment and execution of engineering simulation workflows

on cloud architectures. In *NAFEMS European Conference - Simulation Process and Data Management, Munich* (2018), NAFEMS European Conference.

[134] Martin, J. Managing the data base environment.

[135] Maurer, M., Brandic, I., and Sakellariou, R. Enacting slas in clouds using rules. In *European Conference on Parallel Processing* (2011), Springer.

[136] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017), IEEE, pp. 405–410.

[137] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017), IEEE, pp. 405–410.

[138] Mehlhorn, K., and Sanders, P. *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media, 2008.

[139] Mehra, A. Docker issues: Docker checkpoint an experimental feature, 2019. `https://github.com/checkpoint-restore/criu/issues/718#issuecomment-508638241` (Last visit: 2021-06-17).

[140] Mell, P., Grance, T., et al. The NIST definition of cloud computing.

[141] Menascé, D. A., and Ngo, P. Understanding cloud computing: Experimentation and capacity planning. In *Int. CMG conference* (2009).

[142] Millidge, S. Cloud myth: Ahead of time compilation will save you money, 2020. `https://blog.payara.fish/java-ahead-of-time-native-compilation-on-cloud-economic-myths` (Last visit: 2021-06-13).

[143] Mironov, I., Segev, G., and Shahaf, I. Strengthening the security of encrypted databases: Non-transitive joins. In *Theory of Cryptography Conference* (2017), Springer, pp. 631–661.

[144] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[145] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[146] Murphy-Hill, E., Sadowski, C., Head, A., Daughtry, J., Macvean, A., Jaspan, C., and Winter, C. Discovering api usability problems at scale. In *Proceedings of the 2nd International Workshop on API Usage and Evolution* (2018), pp. 14–17.

[147] Nadi, S., Krüger, S., Mezini, M., and Bodden, E. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 935–946.

[148] Naveed, M., Kamara, S., and Wright, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 644–655.

[149] Networks, S. Skyhigh networks. Online, 2014. `https://www.skyhighnetworks.com/cloud-encryption/`.

[150] NIST. Cve-2016-5195 detail, 2016. `https://nvd.nist.gov/vuln/detail/CVE-2016-5195` (Last visit: 2021-06-17).

[151] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.

[152] of Edinburgh, T. U. The three states of information, 2021.

[153] of Health, U. D., and Services, H. Breach Portal: Notice to the secretary of hhs breach of unsecured protected health information. https://ocrportal.hhs.gov/ocr/breach/breach_report.jsf. Accessed: 2019-04-13.

[154] of Health, U. D., Services, H., et al. Hitech act enforcement interim final rule. *US Department of* (2009).

[155] Oliveira, D., Ogasawara, E., Ocaña, K., Baião, F., and Mattoso, M. An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation: Practice and Experience 24*, 13 (2012).

[156] O'Neil, E. J. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1351–1356.

[157] OpenStack, I. Openstack. *Apache Licence 2* (2021), 86.

[158] Oracle Corporation. Graalvm.

[159] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques* (1999), Springer, pp. 223–238.

[160] Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S. G., George, W., Keromytis, A., and Bellovin, S. Blind seer: A scalable private dbms. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 359–374.

[161] Pemberton, N., Schleier-Smith, J., and Gonzalez, J. E. The restless cloud. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2021), pp. 49–57.

[162] Peter Olson. BigInteger.js. `https://github.com/peterolson/BigInteger.js/`. [Last visited on May 18, 2020].

[163] Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoc, D. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2016), WWW '16, International World Wide Web Conferences Steering Committee, pp. 263–273.

[164] Poddar, R., Boelter, T., and Popa, R. A. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive 2016* (2016), 591.

[165] Poh, G. S., Chin, J.-J., Yau, W.-C., Choo, K.-K. R., and Mohamad, M. S. Searchable symmetric encryption: designs and challenges. *ACM Computing Surveys (CSUR) 50*, 3 (2017), 40.

[166] Popa, R. A., Redfield, C. M., Zeldovich, N., and Balakrishnan, H. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[167] Qasha, R., Cała, J., and Watson, P. A framework for scientific workflow reproducibility in the cloud. In *e-Science (e-Science), 2016 IEEE 12th International Conference on* (2016), IEEE, pp. 81–90.

[168] Rafique, A., Van Landuyt, D., Beni, E. H., Lagaisse, B., and Joosen, W. Cryptdice: Distributed data protection system for secure cloud data storage and computation. *Information Systems 96* (2021), 101671.

[169] Regulation, G. D. P. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46. *Official Journal of the European Union (OJ) 59*, 1-88 (2016), 294.

[170] Research, M. Azure functions public dataset, 2020. `https://github.com/Azure/AzurePublicDataset` (Last visit: 2021-06-13).

[171] Roman, M., Kon, F., Campbell, R. H., et al. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems* (2001).

[172] Russinovich, M. Introducing azure confidential computing. *Seattle, WA: Microsoft* (2017).

[173] Samarati, P., and Di Vimercati, S. D. C. Data protection in outsourcing scenarios: Issues and directions. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), pp. 1–14.

[174] Santana-Perez, I., da Silva, R. F., Rynge, M., Deelman, E., Pérez-Hernández, M. S., and Corcho, O. Reproducibility of execution environments in computational science using semantics and clouds. *Future Generation Computer Systems 67* (2017), 354–367.

[175] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM 64*, 5 (2021), 76–84.

[176] Seacord, R. C. Replaceable components and the service provider interface. In *International Conference on COTS-Based Software Systems* (2002), Springer, pp. 222–233.

[177] Shafiei, H., Khonsari, A., and Mousavi, P. Serverless computing: A survey of opportunities, challenges and applications. *arXiv preprint arXiv:1911.01296* (2019).

[178] Shafieinejad, M., Gupta, S., Liu, J. Y., Karabina, K., and Kerschbaum, F. Equi-joins over encrypted data for series of queries. *arXiv preprint arXiv:2103.05792* (2021).

[179] Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 205–218.

[180] Silva, P., Fireman, D., and Pereira, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[181] Singh, S., and Singh, N. Containers & docker: Emerging roles & future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)* (2016), IEEE, pp. 804–807.

[182] Singhvi, A., Houck, K., Balasubramanian, A., Shaikh, M. D., Venkataraman, S., and Akella, A. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849* (2019).

[183] Smith, B. C. *Procedural reflection in programming languages.* PhD thesis, Massachusetts Institute of Technology, 1982.

[184] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 275–287.

[185] Solutions, N. Optimus workflow engine, 2021.

[186] Song, D. X., Wagner, D., and Perrig, A. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on* (2000), IEEE, pp. 44–55.

[187] Spring. Getting Started | Building an Application with Spring Boot.

[188] Srirama, S. N., and Viil, J. Migrating scientific workflows to the cloud: through graph-partitioning, scheduling and peer-to-peer data sharing. IEEE.

[189] Stankovski, J., Erdeljan, A., Kovačević, I., and Dalčeković, N. Computing data encrypted by paillier encryption scheme using cassandra database.

[190] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on*

Applications, Technologies, Architectures, and Protocols for Computer Communications (New York, NY, USA, 2001), SIGCOMM '01, Association for Computing Machinery, p. 149–160.

[191] Teles, T., and Peters, C. Implementing a crypto services strategy at abn amro bank, 2020. `https://rwc.iacr.org/2020/slides/Peters-Teles.pdf` (Last visit: 2021-06-17).

[192] Thommes, M. Squeezing the milliseconds: How to make serverless platforms blazing fast!, 2017.

[193] Thorne, B., et al. Javallier, 2017. `https://github.com/n1analytics/javallier` (Last visit: 2019-05-15).

[194] Tornow, D. Imperative vs declarative, 2018.

[195] TOSCA-OASIS. Topology and orchestration specification for cloud applications primer v1, 2013.

[196] Wang, L., Duan, R., Li, X., Lu, S., Hung, T., Calheiros, R. N., and Buyya, R. An iterative optimization framework for adaptive workflow management in computational clouds. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on* (2013), IEEE, pp. 1049–1056.

[197] Wang, W., Zhang, L., Guo, D., Wu, S., Cui, H., and Bi, F. Reg: An ultra-lightweight container that maximizes memory sharing and minimizes the runtime environment. In *2019 IEEE International Conference on Web Services (ICWS)* (2019), pp. 76–82.

[198] Wang, X., Feng, D., Lai, X., and Yu, H. Collisions for hash functions md4, md5, haval-128 and ripemd. *IACR Cryptol. ePrint Arch. 2004* (2004), 199.

[199] Wardrop, M. Container security: Theory & practice at netflix, 2019. `https://www.youtube.com/watch?v=bWXne3jRTf0` (Last visit: 2021-06-17).

[200] Weck, W. Independently extensible component frameworks. *Special Issues in Object-Oriented Programming, M. Mühlhäuser (ed.), dpunkt Verlag* (1997), 177–183.

[201] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution.

[202] Weyns, D., Malek, S., and Andersson, J. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7*, 1 (2012), 8.

[203] Williams, D., Lum, B., Koller, R., and Suneja, S. Nabla containers: a new approach to container isolation, 2021. `https://nabla-containers.github.io` (Last visit: 2021-06-17).

[204] Wright, A., and Andrews, H. Json schema: A media type for describing json documents. Internet-Draft draft-handrews-json-schema-01, IETF Secretariat, March 2018. `http://www.ietf.org/internet-drafts/draft-handrews-json-schema-01.txt`.

[205] Xiong, P., Pu, C., Zhu, X., and Griffith, R. vperfguard: an automated model-driven framework for application performance diagnosis in consolidated cloud environments. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (2013), ACM, pp. 271–282.

[206] Yang, P. System design: Consistent hashing, 2020. `https://medium.com/must-know-computer-science/system-design-consistent-hashing-f66fa9b75f3f` (Last visit: 2021-06-17).

[207] Yuan, X., Guo, Y., Wang, X., Wang, C., Li, B., and Jia, X. Enckv: An encrypted key-value store with rich queries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), pp. 423–435.

[208] Zhao, Y., Fei, X., Raicu, I., and Lu, S. Opportunities and challenges in running scientific workflows on the cloud. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on* (2011), IEEE, pp. 455–462.

List of publications

Journal articles

- Rafique, Ansar, Van Landuyt, Dimitri, Heydari Beni, E., Lagaisse, Bert, and Joosen, Wouter. "CryptDICE: Distributed Data Protection System for Secure Cloud Data Storage and Computation." *Journal of Information Systems (2021).*
- Heydari Beni, E., B. Lagaisse, W. Joosen. "Infracomposer: Policy-Driven Adaptive and Reflective Middleware for the Cloudification of Simulation & Optimization Workflows." *Journal of Systems Architecture (2019)* 95: 36–46.

International conference

- Heydari Beni, Emad, Jordy Dieltjes, Eddy Truyen, Bert Lagaisse, and Wouter Joosen. "Reducing Cold Starts during Elastic Scaling of Containers in Kubernetes." *In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC) (2021).*
- Truyen, Eddy, André Jacobs, Stef Verreydt, Heydari Beni, Emad, Bert Lagaisse, and Wouter Joosen. "Feasibility of Container Orchestration for Adaptive Performance Isolation in Multi-Tenant SaaS Applications." *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC) (2020).*
- Heydari Beni, E., B. Lagaisse, W. Joosen, and A. Aly. "DataBlinder: A Distributed Data Protection Middleware Supporting Search and Computation on Encrypted Data." *Proceedings of the 20th International Middleware Conference Industrial Track (2019).*

- Heydari Beni E., Lagaisse B., Zhang R., De Cock D., Beato F., Joosen W. A Voucher-Based Security Middleware for Secure Business Process Outsourcing. In: Bodden E., Payer M., Athanasopoulos E. (eds) *Engineering Secure Software and Systems. ESSoS (2017).* Lecture Notes in Computer Science, vol 10379. Springer, Cham.

International workshops

- Verreydt, S., E. Heydari Beni, E. Truyen, and B. Lagaisse. "Leveraging Kubernetes for Adaptive and Cost-Efficient Resource Management." *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds (WOC'19) (2019).*
- Kaminski, M., Truyen, E., Heydari Beni, E., Lagaisse, B., Joosen, W.. A framework for black-box SLO tuning of multi-tenant applications in Kubernetes. *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds (WOC'19) (2019).*
- Heydari Beni, E., B. Lagaisse, W. Joosen. "Adaptive and Reflective Middleware for the Cloudification of Simulation & Optimization Workflows." *Proceedings of the 16th International Workshop on Adaptive and Reflective Middleware (ARM) (2017).*
- Heydari Beni, E., B. Lagaisse, and W. Joosen. "WF-Interop: Adaptive and Reflective Rest Interfaces for Interoperability between Workflow Engines." *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware (ARM) (2015).*

Engineering conferences and scientific outreach

- Panzeri, Marco, Roberto d'Ippolito, Emad Heydari Beni, Bert Lagaisse, Martin Motzer, and Franz Stöckl. "Smart Deployment and Execution of Engineering Simulation Workflows on Cloud Architectures." *European Conference - Simulation Process and Data Management. NAFEMS. (2018)*
- Makki, M., E. Heydari Beni, D. Van Landuyt, and B. Lagaisse. "Supporting Interoperability and Scalable Customization for Business-Process-as-a-Service." *KU Leuven: Scientific Outreach. (2017)* https://lirias.kuleuven.be/retrieve/502396.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
emad.heydaribeni@cs.kuleuven.be
https://distrinet.cs.kuleuven.be/