Universidade do Porto • Faculdade de Engenharia

Engineering Software for the Cloud
A Pattern Language

Tiago Boldt Sousa

May 2020

Scientific Supervision by
Hugo Sereno Ferreira, Assistant Professor
Filipe Correia, Assistant Professor

In partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Informatics Engineering
by the
Doctoral Program in Informatics Engineering, FEUP

Contact Information:

Tiago Boldt Pereira de Sousa
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Tel.: +351 22 508 1400
Email: tiagoboldt@gmail.com

This thesis was typeset on an Apple® MacBook® Pro running Mac OS® using a Vim-based editor and the free XeTeX typesetting system. The style is based on the one created by Hugo Ferreira for this own Ph.D. dissertation.

"Engineering Software for the Cloud: A Pattern Language"

Abstract

The last decade brought exponential growth in the number of active users on the Internet. Web application giants such as Google, Amazon, and Facebook achieved a user base of billions. Software and their teams began to operate their web applications at an unprecedented scale, vastly enabled by the introduction of cloud computing by Amazon introduced in 2006, revolutionizing how professionals developed and operated their software. The dynamic allocation of hardware facilitated (ultra-)large scale applications to be attainable by any team, mostly due to the shift between CAPEX (the capital cost of ownership of traditional data centers) to OPEX (the operational cost of paying on-demand). However, along with new possibilities, cloud computing introduced new development challenges: lack of broad expertise, novel architectures, new security threats, new models of governance, and dynamic scalability are just some of the most impacting ones. Recent surveys still evidence the lack of expertise as the principal challenge for cloud development teams [Rig19].

In this dissertation, we research how engineers address the intricacies of designing software for the cloud. Documented practices in this domain fall in a wide range of scientific validity, with most originating from limited observation or experience, lacking empirical validation and thus being insufficient in supporting informed design decisions. We set ourselves to identify and document successful design practices for cloud software in a way that professionals can easily employ. With such a goal in mind, we claim that:

> *While engineering software for the cloud, there are categories of recurring problems, which solutions converge from* good design *principles, that adjust to the context where they emerge. Their adoption is a consequence of (1) the awareness a team has of a problem, (2) the characteristics of the product and the company, and (3) the way these solutions relate amongst themselves.*

A preliminary study describes a reference cloud architecture for a research project that supports Ambient Assisted Living. From that experience, we move to study cloud-related practices and tools with a systematic interview of 25 Portuguese startups. This initial

exploration bootstrapped a catalog of potential patterns, which we empirically assess within a local startup, measuring the team's performance before and after adopting the pattern catalog. We observe improvements in operations, configurations management, and build error frequency. This preliminary study provides *plausibility* to pursue our hypothesis further.

We follow extensive literature research and experimentation to *mine* ten novel cloud patterns, and document them, inspired by Gamma et al. [Gam+94], and the subsequent Software Engineering pattern community. The novel patterns introduced in this dissertation are: ORCHESTRATION MANAGER, CONTAINERIZATION, JOB SCHEDULER, AUTOMATED RECOVERY, FAILURE INJECTION, EXTERNAL MONITOR, LOG AGGREGATION, PREEMPTIVE LOGGING, SERVICE DISCOVERY, and MESSAGING SYSTEM. We relate them into a pattern language, along with two practices already well described in the literature: INFRASTRUCTURE AS CODE and AUTOMATED SCALABILITY.

To understand how technology companies adopt these patterns, we employed two complementary empirical methods in industrial scenarios. The first is a case study with five local startups, relating their practices with the ones proposed in the pattern language. During this exercise, we were able to identify additional details used to improve the pattern language. We also hypothesized that these patterns gain relevance for professionals as their product and company matures, and would expect to observe the more mature companies implementing an increased number of patterns. The second was a survey inquiring over 100 professionals about their cloud practices while querying their usage (either directly or indirectly) of the identified patterns. We relate three variables with the number of patterns adopted: product operation's strategy, the number of active monthly users, and company size. We conclude that for the three variables, the average pattern adoption increases with increased maturity in that variable, with particular relevance in the number of monthly active users, the only variable providing relevant statistical results.

Main future work will (1) expand the pattern language, (2) repeat our experimental research with a broader population, and (3) perform controlled experiments to further understand the impact of applying this pattern language.

Resumo

Durante a última década verificou-se um crescimento exponencial no número de utilizadores ativos na Internet. Gigantes da web como Google, Amazon e Facebook atingiram milhares de milhões de utilizadores. Potenciadas pela computação na nuvem (cloud) introduzida pela Amazon em 2006, as equipas de software conseguem atingir uma escala sem precedentes. A alocação dinâmica de hardware capacita o desenvolvimento de aplicações em grande escala, principalmente devido à mudança do paradigma financeiro de CAPEX (o custo de propriedade dos data centers tradicionais) para OPEX (o custo de utilização de hardware como serviço). Juntamente com as vantagens, a introdução da cloud trouxe novos desafios de engenharia, como a falta de experiência, introdução de novas arquiteturas, novas ameaças de segurança, modelos diferentes de operação e escalabilidade dinâmica, são alguns dos exemplos com maior impacto. A literature mostra que a falta de recursos e de conhecimento são em 2019 dos maiores desafios para as equipas de desenvolvimento [Rig19].

Nesta dissertação, investigamos como os engenheiros abordam a complexidade do design de software para a cloud. De igual modo, demonstramos que as práticas documentadas não apresentam validade científica relevante, sendo que a maioria é produto da observação ou de experiência limitada, portanto, incapaz de apoiar a decisão informada no desenho de software.

Neste sentido, pretendemos identificar e documentar boas práticas de design de software para a cloud, de forma a serem facilmente aplicadas por profissionais. Com hipótese principal, propomos demonstrar que:

> *Ao desenhar software para a cloud, existem categorias de problemas recorrentes, para os quais os profissionais convergem para soluções baseadas em bons princípios de desenho, adaptadas ao contexto onde são observadas. A adoção destas soluções resulta da (1) consciência da equipa sobre a existência do problema, (2) das características do produto e da equipa, e (3) da forma como estas soluções se relacionam entre si.*

Num estudo preliminar, desenhamos uma arquitetura de referência para um projeto de investigação no domínio de Ambient Assisted Living. Com base nessa experiência, começamos a investigar práticas e ferramentas relacionadas com a cloud, aplicando entrevistas sistemáticas a 25 startups portuguesas. Esta investigação permitiu criar um catálogo de padrões que posteriormente avaliamos junto de outra startups. Como resultado, foram observadas melhorias nas operações, na gestão de configurações e menor frequência de erros de compilação dessa startup. Este estudo preliminar fornece *credibilidade* para continuarmos a investigar a hipótese apresentada.

De seguida, através de pesquisa bibliográfica e experimentação de tecnologias, capturamos padrões de desenho de software para a cloud, baseadas no trabalho de Gamma et al [Gam+94] e subsequentes contribuições da comunidade de padrões. Os novos padrões introduzidos nesta dissertação são: ORCHESTRATION MANAGER, CONTAINERIZATION, JOB SCHEDULER, AUTOMATED RECOVERY, FAILURE INJECTION, EXTERNAL MONITOR, LOG AGGREGATION, PREEMPTIVE LOGGING, SERVICE DISCOVERY e MESSAGING SYSTEM. Estes são relacionados numa linguagem de padrões, juntamente com duas práticas previamente descritas na literatura: INFRASTRUCTURE AS CODE e AUTOMATED SCALABILITY.

Para entender como as empresas adotam estes padrões, realizamos dois estudos empíricos complementares em ambientes industriais. O primeiro é um estudo de caso com cinco startups, que relaciona as suas práticas com as que são propostas na linguagem de padrões. Durante este estudo, conseguimos identificar novos detalhes de implementação que nos permitiram melhorar a linguagem dos padrões. Nesta fase, levantamos a hipótese de que número de padrões adoptado tende a crescer à medida que os produto e empresa amadurecem. Com esta premissa, esperamos observar que empresas mais experientes implementem um maior número de padrões.

De seguida fizemos um survey com o objetivo de recolher informação de 100 profissionais sobre a utilização das práticas da linguagem de padrões nos seus produtos. Para tal, relacionamos três variáveis com o número de padrões adotados: a estratégia de operação do produto, o número de utilizadores ativos mensalmente e o tamanho da empresa. Concluímos que, para as três variáveis, o número médio de padrões adotados aumenta com o aumento da maturidade dessa variável, com particular relevância no número de utilizadores ativos mensalmente, a única variável que fornece resultados estatísticos relevantes.

Como trabalho futuro destacamos a importância de (1) expandir a linguagem de padrões, (2) repetir o caso de estudo com maior amostra, e (3) realizar experiências controladas para avaliar o impacto da utilização da linguagem de padrões.

Contents

List of Figures

List of Tables

Foreword by João Azevedo

Over the last decade, the way most software applications are delivered to end-users changed significantly. The ubiquity of internet access shifted the paradigm from desktop applications to web-based applications. This has made software applications more accessible, data more readily available, and users more connected. This shift, allowed by the widespread internet access, liberated engineers from challenges related to developing for multiple operating systems, packaging, and distributing updates. It has, however, introduced new challenges. There are higher security risks when using web-based applications, internet connectivity impacts the experience, and maintenance fees are higher, just to name a few. Cloud computing, introduced by Amazon Web Services, democratized access to computing resources. While the idea of not having to physically manage the required resources to run an application is compelling, it doesn't come without implications to the ways we design software.

As software engineers designing and developing applications that take advantage of cloud computing, we are faced with problems orthogonal to the application domain. How do we run our applications in an environment that we don't manage? How do we ensure that such an environment provides the resources we need? How do we make sure that applications are running as expected? How do we handle unexpected loads to our application that is now exposed to the web? The vast majority of these questions were answered by multiple people, often in a similar fashion.

This thesis delivers solutions to common problems when designing and developing software that runs or interacts with the cloud. It does so by identifying and documenting patterns applied by the industry to solve common cloud-related issues. It also correlates the adoption of these patterns with different software company categories, both validating their relevance and applicability in different scenarios. This work is valuable for software engineers venturing into cloud development for the first time, due to the solutions presented here for problems that they're likely to encounter. Experienced cloud practitioners will also value it due to the case studies that showcase the usefulness and applicability of the different patterns for different scenarios.

I was fortunate to work with Tiago while he was writing this thesis, and we were collectively struggling, as a company, with many of the issues whose possible solutions are presented here. I, therefore, recommend this work to all software engineers working with cloud solutions, regardless of their experience with it.

Preface

I often ask colleagues what drove them to become IT professionals, only to find the answer is a cliché. I enjoyed playing video games as a kid, most say, so I went ahead and tried to understand how the machine worked as well. My story was no different. I remember the 286 PC with DOS 5 at my grandparent's house, around 1993, which my uncle used for his terminal-based spreadsheets, with Lotus, I believe. I remember thinking how incredible it was for it to be able to recalculate everything once a cell was changed. But, of course, there was one thing that impressed me far more than the spreadsheets. Gorillas, a game distributed with QBASIC as a demo for the programming language, hooked me to that machine. Two gorillas had to throw a banana at each other, in turns, asking the player to input the angle and speed of the throw. When you managed to hit the other, an incredible explosion (by the standards at the time) would destroy your opponent. Browsing images of the games today still brings me great memories.

By 1995, computers became a more affordable piece of technology and entertainment. My enthusiastic wishing for one eventually led to my parents to acquire a Compaq Presario 486 PC, 8MB of RAM, 640 MB of disk space and the at-the-time disrupting compact disk reader, running Windows 3.1. A free upgrade to Windows 95 would soon follow by mail in a Compact Disk. I quickly managed to get some friends and family to lend me some games in floppy disks, but I had no idea how to start them. However, I had an MS-DOS user manual, so I experimented with the commands in the book and learned what they did by experience. They were obviously in English, which I knew little at the time, so I had to do it with a dictionary next to the manual or keep bothering my mother for help. Along the way, I often deleted critical files from the file system, so next in the learning process was how to reinstall the operative system, which became a rock star-like skill with friends and family. A few years later, a friend lent me a photocopied manual for Visual Basic 6, which I used to learn about programming and building little programs, like visual calculators or simple games.

There was, though, one thing while using computers and consoles that I always felt lacking. They were much more fun when shared than when used by myself. The late

'90s introduced me to the game-changing Internet. My first modem came inside the Sega Dreamcast console. It had a browser, a keyboard, and it enabled accessing online chat groups. During those early days, Internet access was limited in time and speed. Despite this limitation, it seemed possible to read about anything and reach anyone on a global scale. That felt incredible. As soon as I got an updated desktop computer, my Internet time expanded, and I used it to further my computer knowledge, which at the time meant building computer scripts and hacking into friends' computers. How fun it was to have a friend in a panic saying that their CD drive kept ejecting. The pleasure in it came not from having control over a friend, but from having control over the machine.

At the age of 16, I became interested in Linux and managed to have a local cyber-cafe owner teach me about it and get a dual boot at home soon after I compiled my kernel with support for my graphics card to play games with it. I was already an engineer at the time, by definition, so it was only natural to pursue a Masters in Informatics Engineering.

I was always inclined to learn more about how people could cooperate using computers. Web 2.0 was happening in full effect during my Masters, and Internet reach was expanding at an incredible pace. I was fortunate to do my Master's dissertation on HTML5 and how it enabled a new type of interactivity between users and web applications, which furthered my curiosity about the architecture of web applications at scale.

Right after graduation, and a short trip through the industry as a contractor for ShiftForward, my supervisor, Hugo, challenged me to go back to academia and work on *cool stuff*. Whatever *cool stuff* interested me, as long as I was willing to work for it, he would sponsor a Ph.D. proposal. The idea pleased me, so I got a scholarship with INESC TEC to help them implement the infrastructure for a research project. The initial experimentation with large scale architectures furthered my curiosity on the subject. With Cloud Computing gaining traction, it became clear that I had found my *cool stuff* subject.

I set myself to research best practices to design software for the cloud, which led me to today, dear reader, and the completion of this dissertation. I hope you find this work meaningful for personal use. Feel free to share it with your peers. I believe its knowledge, while respecting academic research guidelines, will be readily applicable in the industry and help professionals design, improve, or validate their cloud architectures.

This dissertation would not have happened without people I hold dear supporting me along the way, to whom I am immensely thankful. First and foremost, I dedicate this work to my parents, who supported my curiosity for computers during my early years and were encouraging me towards a Ph.D. since before I finished my Masters. To Mariana, who (tried to) kept me sane by ensuring I had a life outside this research along the way. To João and Sandra, for making dinner plans most weeks, allowing me to forget work for

some hours. To Hugo, who some (many) years ago challenged me into this path, and was always available to discuss *cool stuff*, as far as it was after 14:00. To the team at Velocidi, with whom I've been cooperating since 2011, for being incredible engineers, discussing, and putting into practice most of the knowledge gathered in this dissertation, and to Paulo Cunha, who taught me much about running a business. To Ademar Aguiar, who co-supervised the early stages of this research and to Filipe Correira, who co-supervised the later stages of this research. To Lina, Marisa, Sandra, and Pedro, the staff from the Informatics Engineering Department from FEUP, for guiding me through the intricacies of academic bureaucracy. Finally, to so many other colleagues, friends, and students, who along the way positively influenced this research or my life: André Cardoso, André Restivo, André Silva, Bruno Lima, Carlos Teixeira, Diana Norinho, Diogo Guimarães, Diogo Sousa, Francisco Almeida, João Azevedo, João Costa, João Pascoal Faria, João Pedro Dias, João Pedro Pereira, João de Almeida, Luís Ferreira, Miguel Montenegro, Ruben Barros, Rui Gonçalves, Zé Miguel Neves, the PLoP community, and so many others.

Chapter 1

Introduction

Access to the Internet became a right in civilized countries [Uni03], enabling ultra-fast communication and collaboration, facilitating a new generation of web-based applications. These applications introduce requirements unprecedented to software engineering, such as resiliency and scalability, ensuring that they are always available even under variable demand. Cloud computing provided resources that enable developers to cope with these requirements but demands a paradigm shift from their traditional approach to software development and, mainly, operation. This work researches how cloud software is engineered, from design to operation. We identify a lack of empirically validated knowledge in this domain and contribute to it, with a set of ten solutions for recurrent problems (to which we call patterns), that we relate in a pattern language. We validate the impact of this knowledge via a case study with five startup companies and a survey with over 100 responses. This research enables cloud engineers to make informed decisions while designing their cloud software.

1.1 Context

The Internet is strongly influencing how we live our lives. The World Wide Web (WWW) enables us to buy groceries, stream TV shows, and have all our colleagues, friends, and family accessible on the other side of the smartphone that we carry on our pocket. Over 53% of the world's population is online daily. These individuals are potential users for any online businesses, making the WWW an appealing channel for businesses to reach their target audience. That motivation resulted in the creation of over two billion websites [Int19].

Over the years, websites became increasingly more complex, ranging from static HTML sites to highly complex browser-based applications, such as Google Drive or Facebook [And16]. They have also grown to manage large volumes of data for tens of millions of users. Along the way, infrastructure and technologies had to adapt to accommodate the increased complexity, motivating the birth of cloud computing.

While targeting a global market, development teams had to worry about scaling their hardware and software. Scaling infrastructure required a preemptive investment in hardware, which could be a prohibitive cost to some companies. Without the proper preparation, traffic spikes could overwhelm the infrastructure supporting the application, rendering it unresponsive, an unacceptable scenario, since users do not cope well with failing software and quickly jump to the next available alternative. With the global reach of web applications, and with demand varying during the day, dynamically scaling the software became a latent necessity.

To address this opportunity, Amazon introduced the notion of cloud computing in 2006 [ZCB10]. Cloud deprecated the need to invest in infrastructure for operating software over the Internet. Instead, it introduced a service model where software, platforms, or infrastructure could be allocated on-demand, as a service, on a pay-per-use basis, working as a commodity, just like electricity [Ban+11; TCB14]. Cloud providers achieved efficiency through an economy of scale, benefiting both the provider and customer [Sav11]. This approach enabled small software companies and individuals to become competitive at building software for a global market. This was partially possible due to the ability to dynamically scale their system, considering the current traffic generated by their users at any given point in time [Bel+11].

Cloud empowered developers to operate their software at a larger scale while introducing the challenge of operating at such a scale. New architectures, frameworks, and tools were required for adopting this new paradigm.

1.2 Motivation

Software engineering is one of the fastest expanding branches of engineering, further motivated by the widespread of the Internet and the explosive growth of software businesses built on top of it. The demand for new engineers is growing at a higher rate than the pace at which they are graduating [Taf15]. Not only the human resources available are scarce, but there is also a lack of high-quality support materials on how to develop for the cloud [Rig19].

Given the limited human resources and lack of reliable knowledge sources, it is not trivial for engineers to make informed decisions while building software. To facilitate their job, we need to understand the complexity of the problems they most often address. What recurrent problems can we observe while developing software for the cloud? Are there key characteristics that influence when problems emerge? What strategies can we adopt to solve them? How do the problems vary with the context, and how can we adapt to it? How can we capture this knowledge and make it readily available to other cloud engineers? Are companies even aware that these problems exist?

Cloud-related architectures, technologies, and overall knowledge have grown to a proportion that became challenging to navigate. There is an evident lack of research supporting architectures for cloud software [Rig19], namely, identifying what forces drive successful cloud software, and the guidelines to optimize them. While some authors are working towards them, these are rarely supported by scientific evidence.

With this work, we propose to research the problem of designing software for the cloud, deprecating the need for months of investment in Research and Development (R&D), or preventing sub-optimal decisions while developing cloud products. We want to provide developers with empirically validated knowledge that will help them make informed decisions for their cloud architectures.

1.3 Engineering Software for the Cloud

Cloud adoption is growing, but not without its engineering challenges, as further elaborated in Chapter 3 (p. 25). Software engineers are tasked with the challenge of designing software for the cloud and ensure its success. Thinking about it, how do we decide how to designing software? What makes it a complex task? Can we distinguish good and bad design? What influences our implementations?

To address these questions, let us start by understanding what design is in the context of software. The Merriam-Webster dictionary defines design as 1) *"a particular purpose or*

intention held in view by an individual or group", 2) *"a plan or protocol for carrying out or accomplishing something"*, and 3) *"an underlying scheme that governs functioning, developing, or unfolding"* [Mer]. Software design relates to these three descriptions. Engineers have to identify the purpose of the product that they want to build and synthesize a plan to build it by defining the scheme or construct that will support its development.

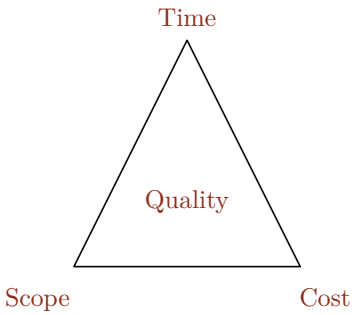


Figure 1.1: The Iron Triangle [Bro15; Atk99] depicts the competing forces that software engineers need to balance while developing their software: time, cost, quality, and scope. Changing one will always result in an impact on the others.

The development process is driven by the iron triangle[1] forces: time, cost, and scope, and how they influence quality [Bro15; Atk99], depicted in Figure 1.1 (p. 4). These need to be balanced appropriately for a successful outcome. For example, to achieve the same result in less time, one either has to increase resources (cost) or accept an impact on quality.

We believe that, just like with the iron triangle, a good design positively influences the development phase and enables a sustainable product. A correlation between good design and resulting software quality has been identified [MT10] and demonstrated by multiple authors [BBM96; BM96]. The right design will result in higher software quality, implemented in less time, with the same costs. Yoder and Foote claimed that *if you think good architecture is expensive, try bad architecture* [FY99], referring to the fact that a bad design can result in damaging costs to a software project.

A challenge with software design is that it is not an exact science. There is no golden rule to solve all problems. The same solution might not be a viable solution to a similar problem in a different context or one that needs to balance forces differently. Still, there are invariant qualities that will positively influence cloud software from its development to operations, as it is the case with testability, scalability, extensibility, amongst others.

[1] The iron triangle is a triangle because the initial version only considered time, cost, and quality as forces. Several variations have been introduced since, with the scope being commonly accepted as the fourth force.

These are essential requirements for relevant cloud-related practices such as Continuous Integration (CI)/Continuous Deployment (CD) or to build elastic applications[2].

1.4 Patterns and Pattern Languages

Most often than not, software engineers are designing solutions to problems that others have already worked on, benefiting from existing knowledge instead of having to *reinvent the wheel* [Boo04]. As discussed in the previous section, the right design knowledge can help engineers ensure software quality without impacting time or cost, resulting in improved development efficiency for a given scope. These designs tend to naturally emerge in multiple and independent approaches to the same problem and often share the same qualities.

Christopher Alexander addressed the problem of capturing recurring problems and their solutions in the context of civil architecture in the 1970s. In his work *A Pattern Language*, Alexander struggled with the need to document and share architectural knowledge, which could be easily applied by his peers. He came up with the concept of patterns, which he describes as a recurrent problem and its respective solution in a given context [AIS77].

Patterns often provide alternative solutions, adapting to the balance of forces in each specific context. The forces identify the characteristics of the problem and its context and often oppose each other, evidencing why the problem is complex to solve. Finding the solution that balances the forces is necessary to make the solution fit the problem. An example of how forces need to be balanced is related to the iron triangle. It is impossible to create software that is complex and has high quality in a short time and without little investment. So, scope, quality, time, and cost are competing forces that need to be balanced according to a context to find the optimal development strategy for that same context.

Along with patterns, Alexander introduced the concept of pattern languages. Kuhne considers patterns as elements of grammar that, by when related, define a language that tells us how to weave the patterns together [Kuh99]. Pattern languages are then a set of patterns applicable in a specific domain that can be used together to solve related problems.

Later, in his book *The Timeless Way of Building*, Alexander described his pursuit

[2] Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible [HKR13].

for a *quality without a name*, an objective and precise quality that had no single word to describe it. When achieved, the solution *felt right* for the problem. He used several alternative words to describe this fitness, in the absence of one that would sum them up: alive, whole, comfortable, free, exact, egoless, eternal. Such a solution made the architect *feel good* with his work [Ale79]. Alexander went on to argue that patterns and pattern languages were the gateway to achieve this *quality without a name*. The patterns in the pattern language would allow the adopter to design buildings just like words in our language allow us to build sentences. Despite the architecture background, this quality, or concept of ideal design, is common to all creative fields, namely in software engineering. When a solution is just right, it benefits from the *quality without a name*.

The usage of patterns and pattern languages as a way for knowledge sharing has later been adopted by software engineers to document effective software design decisions [Fow06]. *Design Patterns: Elements of Reusable Object-Oriented Software* is the most well-known work, adopting patterns to document Object-Oriented Programming (OOP) design practices [Gam+94]. The book is often adopted to teach OOP design practices in most software engineering degrees. Patterns and pattern languages went on to provide not only knowledge but also vocabulary that adopters can use to argue about their problems and solutions [SFJ96].

1.5 Research Goals and Contributions

With this work, we aim at identifying cloud design practices in the form of a pattern language, facilitating the bootstrap and decision-making process while designing software for the cloud. These patterns are empirically evaluated for their completeness and relevance in the industry. The contributions of this research are:

A review of state of the art for cloud computing. Identifies recurrent challenges for designing software for the cloud and how developers are addressing them. More details in Chapter 3 (p. 25).

Preliminary study. Experiment with cloud technologies and designs to acquire the expertise required to pursue this research.

A pattern language for engineering software for the cloud. Describes design best practices for cloud computing, helping engineers design their applications. More details in Chapter 6 (p. 69)

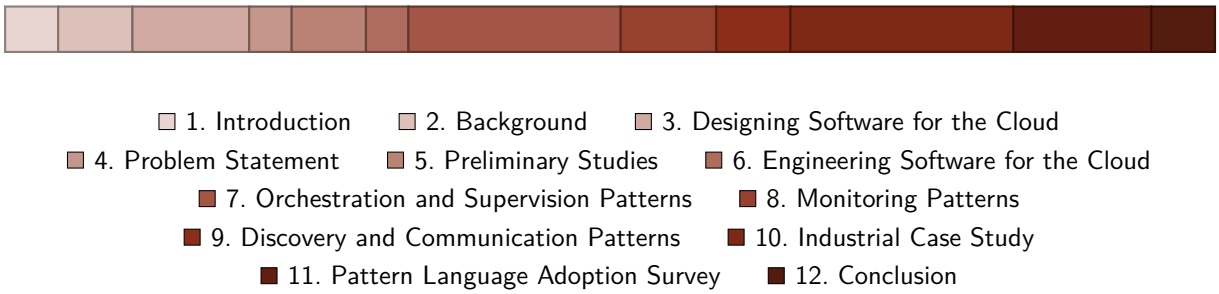


Figure 1.2: Overview of the dissertation's chapters and their relative dimensions.

Industrial case study. Interviews with five startups developing software for the cloud, inquiring about their cloud designs and challenges, relating those to the described pattern language. More details in Chapter 10 (p. 149).

Pattern language adoption survey. Recurs to over one hundred respondents from professional software developers, enabling an evaluation of the individual adoption of each pattern in the industry, and the discussion of how they relate to company characteristics such as company size or operations strategy. More details in Chapter 11 (p. 191).

1.6 How to Read this Dissertation

The target audience for this work are software engineers, designers, architects, developers, or anyone involved in defining the architecture for a cloud application. This research will empower the reader with a set of best practices that can bootstrap their cloud development, significantly reducing the R&D time required to achieve a highly scalable system. We also target this work at cloud researchers, providing them empirical data regarding cloud adoption in the industry, and an extensive discussion relating the captured data to the proposed best practices.

The remaining of this document is organized in eleven chapters. Figure 1.2 (p. 7) provides a visual representation of the relative size of each chapter. These are:

- Chapter 2 (p. 11) describes cloud computing and its ecosystem as a platform upon which scalable applications can be built. It briefly describes DevOps as a cultural movement and how it expedites cloud development. We introduce the context and concepts relevant to read the remaining of this work. Readers familiar with cloud computing can skip the chapter.

- Chapter 3 (p. 25) describes the current state of the art of software development for the cloud. The recurrent challenges and failures are introduced, along with the

research from multiple authors on how to improve cloud development, namely by using design patterns.

- Chapter 4 (p. 47) discusses why the current state of the art provides insufficient material for supporting the development of cloud software and elaborates on the goals of this research.
- Chapter 5 (p. 55) describes two cloud projects we have contributed to, as a strategy to deepen our knowledge on the intricacies of cloud computing development. The first describes the contribution of a reference cloud architecture for an e-health European research project, with concrete scalability and privacy requirements. The second describes preliminary industry research on cloud practices and how these can be formalized and used to improve the cloud development practices of less experienced teams, with a concrete case study.
- Chapter 6 (p. 69) introduces our pattern language and the ten patterns that currently compose it, as well as describing details regarding their implementation.
- Chapter 7 (p. 77) details five patterns for cloud orchestration, supporting the deployment and operation of cloud applications.
- Chapter 8 (p. 117) details three patterns for monitoring the status and state of cloud software, continuously verifying their correct execution or inspecting their state when needed.
- Chapter 9 (p. 135) details the last two patterns in the language, which facilitate service discovery or communication, essential for cooperation between services.
- Chapter 10 (p. 149) describes a case study with five interviews with Portuguese startup companies to evaluate if their cloud architectures have intuitively implemented the patterns from the pattern language, as well as trying to identify possible implementation details that could improve the individual patterns.
- Chapter 11 (p. 191) describes a survey of over 100 companies, evaluating their adoption of the identified patterns. Their pattern adoption is correlated with company size, active users, and product operations strategies in an attempt to identify the factors that could motivate others to apply the patterns.
- Chapter 12 (p. 217) revisits the contributions from this work and proposes possible future research paths for continuing this research.

There are often references to pattern names during this research, which are identified using SMALL CAPS.

Chapter 2

Background

This chapter introduces the topics of World Wide Web (WWW), cloud computing, patterns, and pattern languages, and can be skipped by experts in these subjects. We revisit the WWW has a channel for disseminating content online, which enabled the natural evolution towards web applications. Cloud computing is described as the infrastructure that empowered the WWW revolution. We introduce DevOps as a mindset motivating autonomous teams that automate quality assurance, deployment, and operations, eliminating the segregation of responsibilities and the need for human intervention in software operations. Finally, we describe Design Patterns as a source of knowledge for software designers to optimize their design decisions while minimizing investment in Research and Development (R&D).

2.1 The World Wide Web

The WWW was invented at Conseil Européen pour la Recherche Nucléaire (CERN)[1] in the late 1980s by Sir Tim Berners-Lee. There he built the first web server and browser [And16]. In its original form, it was a collection of documents identified by a Uniform Resource Locator (URL) made available using the Internet. Documents would be either static web pages or files that could be remotely made available using a web server. Albert compared the WWW to an elaborate graph whose vertices are documents and edges are links between them [AJB99].

By that time, making software available to users was a troublesome process. Programmers needed to compile their software for a given number of platforms, to distribute their binaries to the user and for him to then install the software on his computer or other devices. Several issues arise during this process, namely, software incompatibilities across platforms, user permissions, insufficient hardware capabilities, or merely the lack of technical knowledge from the user to install the software.

During 1999, the WWW went through a great evolution which DiNucci called *Web 2.0* [DiN99]. Web 2.0 introduced user-generated content, as opposed to traditional static web sites. Web applications were built using web technologies that use web browsers as clients, requiring only an Internet connection for the interaction between the users and the remote server. Asynchronous JavaScript further improved this experience by allowing interactions with the server without having to reload the web page, resulting in the introduction of interactive and dynamic web pages [Mur07]. During this time, applications were deployed in pre-allocated infrastructure. Companies had to invest in data centers and infrastructure and hire dedicated teams to operate them. The upfront investment was prohibitive for most companies, rendering it very difficult for smaller players to launch a business online [Gre+08]. However, as the Internet was growing[2], it was the ideal channel for disseminating content or supporting businesses. The opportunity for cloud computing emerged when applications had to become elastic and scale horizontally [NS14] to multiple machines. Cloud computing provided services to support the design, implementation, and operation of such horizontally-scaled applications, meeting a demand for agile infrastructure [Deb08] and cost-efficient operations.

[1] CERN is the European Organization for Nuclear Research. http://home.web.cern.ch/
[2] Today, the Internet reaches over 53% of the world population [Int19]

2.2 From SOA to Microservices

Late in the '90s, web software development observed a considerable increase in complexity. A new architecture paradigm was introduced to keep development manageable, called Service Oriented Architecture (SOA) [Nat03], as introduced by Gartner in 1996. SOA proposed the decoupling of complex applications into smaller components, called services, that could be individually implemented and deployed. With the appropriate communication channels, they would be able to cooperate in providing parts of a complex application [Ric15].

As an example, consider a Representational State Transfer (REST) HTTP service that relies on an external authentication service. Different development teams can own the REST HTTP and the authentication services, given that they have agreed on how the services cooperate, typically in the form of an Application Programming Interface (API). SOA's relevance expanded during the 2000s, with the introduction of cloud computing and web services. Initial implementations were mostly based on Simple Object Access Protocol (SOAP) [Ric15], later replaced by the less verbose REST [Fer+13].

During the early 2010s, some characteristics behind SOA begun to be questioned as to their fitness for the agile development most teams were pursuing, as described by Richards [Ric15]. In March 2012, James Lewis introduced *Micro Services - Java, the Unix Way* [Lew12], in what was possibly one of the first references to Microservices. Lewis described the following characteristics for Microservices:

Each application only does one thing. Services must be atomic in their responsibilities.

Small enough to fit in your head. Services are simple enough so that the developers fully understand it.

Small enough that you can throw it away. Services are kept small to the point that a rewrite would be easily attainable.

Microservices enabled the distributed ownership of application components by individual development teams, facilitating how software development and operations could scale [Lew12]. It later led to the DevOps mindset (see Section 2.5 (p. 19)), with developers owning not only the implementation but also the operation of their service. Empowered by cloud computing, development teams had a strategy to build and operate their software at scale.

2.3 Cloud Computing

Amazon Web Services (AWS) introduced cloud computing in 2006 as a set of managed services that provide building blocks for building software [Gar06]. These services were provided as a commodity that could be acquired on a pay-as-needed basis, just like water or electricity [MK10]. Tootsie described the cloud as a new paradigm for providing services on a pay-as-you-go basis. It removed most upfront costs of setting up an IT infrastructure, moving the cost of infrastructure from Capital Expenditures (CAPEX) to Operational Expenditures (OPEX) [Feh+14], while enabling organizations to adjust resources on demand [TCB14]. The name cloud came due to the use of a *cloud* as a representation of the Internet on most architecture diagrams [Gar06]. Cloud services are available at three different service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [TSB10].

With the aforementioned Internet growth, the cloud became a preferred channel on top of which applications are being developed, pushing the cloud market value in 2019 above US$220 billion [Gar19].

2.3.1 Brief History of the Cloud

The demand for computational power has been recognized for decades. By 1967, Irwin predicted that the future would bring computational power at large scale, provided by large data centers to the general public, in a way analogous to the distribution of electricity [Irw67]. Amazon, in 2006, made that idea a reality through the introduction of cloud computing [ZCB10]. AWS, a subsidiary of Amazon, introduced a set of services that revolutionized how software was designed and deployed. Their goal was to provide any software developer with infrastructure and software services on-demand to simplify their software life cycle. Pay-per-use cloud services enabled small companies to build complex products without requiring the funding for building data centers and have the team to operate them.

AWS initially approached the market with three services: Elastic Compute Cloud (EC2) provided on-demand computing infrastructure, Simple Storage Service (S3) a managed file storage, and Simple Queue Service (SQS) a message queue service for enabling service cooperation. The three were controlled programmatically and billed on a per-use basis. Over the years, AWS added dozens of services to their list[3].

Competition to AWS quickly followed. Google Cloud Platform launched its services

[3] Late in 2019 AWS provided 165 different cloud services.

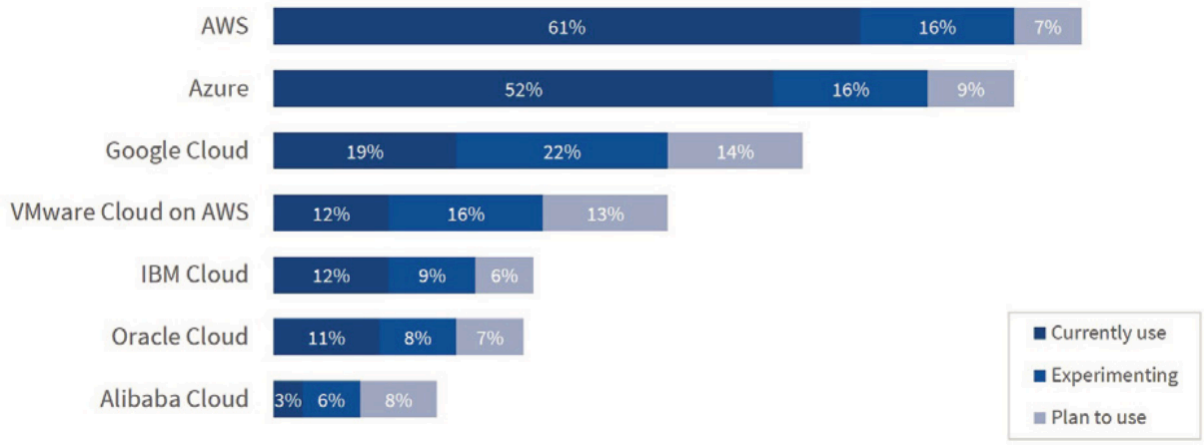


Figure 2.1: Cloud provider market distribution in 2019. Several respondents adopt multiple cloud providers. Credits: RightScale cloud report [Rig19].

in April 2008, and Microsoft Azure arrived in February 2010. Together, they operate most of the cloud market [Rig19], as depicted in Figure 2.1 (p. 15). It is observable that many respondents to RightScale's survey were using multiple cloud providers, a common practice for improved performance and redundancy, as described in Section 2.3.2 (p. 17).

2.3.2 Service Models

Cloud computing enables software development by providing reusable components as a service, often referred to as Everything as a Service (XaaS) [Ban+11]. The components are often categorized into one of three categories of service commercialization: SaaS, PaaS, and IaaS. All three service levels have been described in the National Institute of Standards and Technology (NIST) definition of cloud computing [MG11].

Software as a Service SaaS provides services on demand. A wide range of services are available today, such as databases or email sending services. These services can be adapted to facilitate the development of new services by outsourcing part of its required building blocks into them. Gartner describes it as software that is owned, delivered, and managed remotely by one or more providers [Gar14]. The provider delivers software in a one-to-many model to all contracted customers, on a pay-for-use or subscription-based on use metrics. According to [Gar12], SaaS revenue was forecast to reach $14.5 billion in 2012, a 17.9 percent increase from 2011. SaaS continued to grow steadily, reaching total revenue of $80 billion across all public cloud providers by 2018 [Gar19]. Motivation to adopt SaaS is similar to the motivation from developers to adopt third party libraries while developing software.

It provides reusable components they can adopt, focusing their development efforts on the novelty of their product [Cus10; Gar14].

Platform as a Service PaaS provides a fully managed and scalable running environment for applications built with the supported programming languages and libraries [MG11]. The platform manages all server components, enabling developers to focus only on the application itself. PaaS typically integrates well with the development team's source code management, being able to integrate deployments with their Version Control System (VCS), often Git [Ode14].

Google Cloud Platform was the first large scale provider for PaaS services with their App Engine, released in 2008 [Jac08]. PaaS often provides monitoring dashboards for the allocated environment, showing metrics such as requests per second or allocated machines. The team can often manually change the hardware allocation in these dashboards. Google App engine dashboard is shown in Figure 2.2 (p. 16). PaaS providers often complement their offer with SaaS services that provide components for developers to build their applications with, such as databases or cache services.

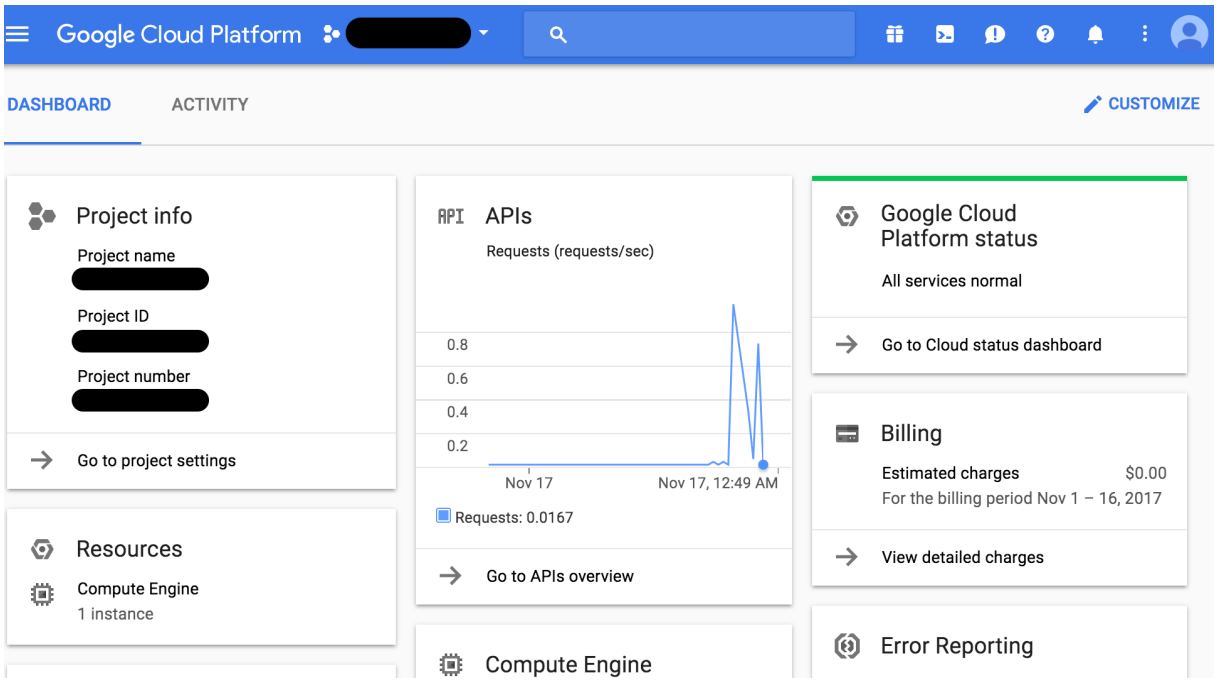


Figure 2.2: Google's App Engine dashboard, showing the incoming requests per second, allocated computing instances, and estimated billing. Account details have been redacted.

Infrastructure as a Service IaaS stands at the lowest level, providing infrastructures such as virtual machines and load balancers, providing the closes environment to

managing the actual hardware. Users are responsible for creating and managing their environment, and then orchestrate their software on top of it. It is the lowest abstraction level of cloud service, and often consists of computing, storage, load balancers, networks, and other hardware (often virtualized [ZCB10]), that the user must configure to adapt to the hosted application.

Multi-cloud Cloud providers have grown to offer hundreds of services, with a geographically redundant availability [Ser19a]. Nevertheless, some applications have particular cloud requirements that cannot be addressed by a single cloud provider. Consider an application that for latency reasons requires being deployed in both China and the Central United States. If we consider AWS and Google Cloud, AWS is present in Beijing, China, but not in Central US, while Google Cloud has a data center in Iowa, but not in China [Ser19a; Goo19]. Such applications can adopt multiple clouds, or multi-cloud, where one or more cloud providers are used while designing the cloud solution.

A common strategy to expand to multi-cloud consists of deploying decoupled or replicated components from an application into different providers [AKL10]. Cooperation between them is achieved by using the Internet and service communication using a broker[4] or APIs [Pau+14]. Chinneck states that for optimal usage, multi-cloud requires automatic infrastructure management [CLW14]. Adopting multi-cloud is not a trivial decision, as it requires the team to prevent vendor-lock in, a natural temptation considering the SaaS offering that each vendor provides, which can decrease development time and complexity [Cus10; Gar14].

Along with public clouds, private clouds can also be adopted for increased privacy and cost efficiency, at the cost of initial hardware investment, physical space allocation, and human resources [MK10].

Grozev et al [GB14] described a taxonomy for multi-cloud architectures and surveyed 20 multi-cloud projects, concluding that these improve Service-level Agreement (SLA) performance and often require a broker to connect them multiple components. The Uni4Cloud project demonstrated how to increase redundancy through the use of multi-cloud by deploying an application into IaaS from two different providers [SM11]. SeaClouds is another EU research project that introduced tools for seamless adaptive multi-cloud management [Bro+14].

[4] A message broker mediates messages between services. It can be implemented with a message queue service such as AWS SQS.

2.4 A Note on Agile Software Development

Previous to the introduction of cloud computing, software development cycles were very long. In the '90s, new software versions were available each 24 to 36 months. The early days of the Internet businesses compressed this window to a much shorter cycle of three to six months [Bas+01]. Companies were built to optimize their efficiency at the cost of flexibility [Pos16]. Cloud computing, pursuing the increased Internet reach from the mid-2000s, and the market competition that came with it, resulted in the demand for software with shorter cycles, reducing the deployment cycle down from years to thousand of deployments per day, as it is the current case of Netflix [DJG18].

Melvin Conway stated that *Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.* [Con]. Just like Conway stated, highly bureaucratic organizations, while possibly efficient, would be highly bureaucratic at building their software, lacking the required flexibility demanded by cloud software. Agile methodologies provided an initial step towards achieving flexibility in software development by valuing individuals and their interaction over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan [Bec+01].

Scrum, one of the emerging agile methodologies from the early 2000s, prescribed self-organizing and cross-functional teams [SS17]. This parted from the segregation of responsibilities over multiple teams such as architects, developers, Quality Assurance (QA), and operations. Self-organizing teams are never blocked by external parties and have all the necessary resources at all times to attain excellent performance. Self-organizing teams have proven to be successful at eliminating inefficiencies and reducing the development cycle time, with companies like Microsoft and Netscape observing a reduction in product development time by 50% [Bas+01].

The Scrum authors recommend the methodology for the operation of cloud environments, as well as the products built for it [SS17]. In the context of cloud computing, when development efficiency increased, the autonomous team had to spend vast amounts of their time handling their operations, which non-exclusively including setting up new infrastructure, deploying the software, or monitoring it, and, at the time, started by being a mostly manual process. The need for automating operations was latent, leading to DevOps.

2.5 DevOps

Debois coined the term DevOps during the 2008 Agile Conference in Toronto. He suggested that agility should be present beyond development, and software teams should incorporate operations as part of their development process as well. DevOps improves on the human-centric operations by leveraging a programmatic approach to automated operations, developed as part of the software development process [Deb08].

DevOps is a mindset for software development that proposes the aggregation of the development, QA, and operations teams into a single team that owns the whole application lifecycle. Professionals in the team might still have different roles, but the team will be autonomous and have a single agenda while engineering and orchestrating their software [EAD14; Día+18; SNP15]. Another strong position of the DevOps movement is that QA and operations tasks need to be fully automated [Roc13]. Automated tests and scripted deployments would replace the manual process of testing and deploying software. This change enables the adoption of Continuous Integration (CI) and Continuous Deployment (CD), with the application being deployed automatically as soon as new code is available and validated by automated tests [Lou12].

Despite its advantages, moving towards DevOps from a traditional software team organization is not trivial. The human factor introduces a natural resistance to change, compromising the transition from independent to multidisciplinary teams [Tec14]. Furthermore, the DevOps technological landscape quickly exploded with hundreds of tools, making it difficult for newcomers to decide on the best tools to adopt.

The XebiaLabs DevOps periodic table, depicted in Figure 2.3 (p. 20), identifies the top 120 technologies into fourteen categories that they consider the most relevant for DevOps implementation [Xeb19a] at the current time. These are but the tip of the DevOps *iceberg*, with an extensive list also from XebiaLabs containing, at the time of writing, 489 entries [Xeb19b]. This extraordinarily complex ecosystem is non-trivial to navigate, resulting in too many open questions for newcomers, which must be addressed in not that much time before their focus is moved to product development.

2.6 Software Design and Design Patterns

The definition of software design is often vague or misinterpreted. To better understand it, we must understand what design is. According to the Merriam-Webster dictionary, design is defined as (1) *a particular purpose or intention held in view by an individual or group*,

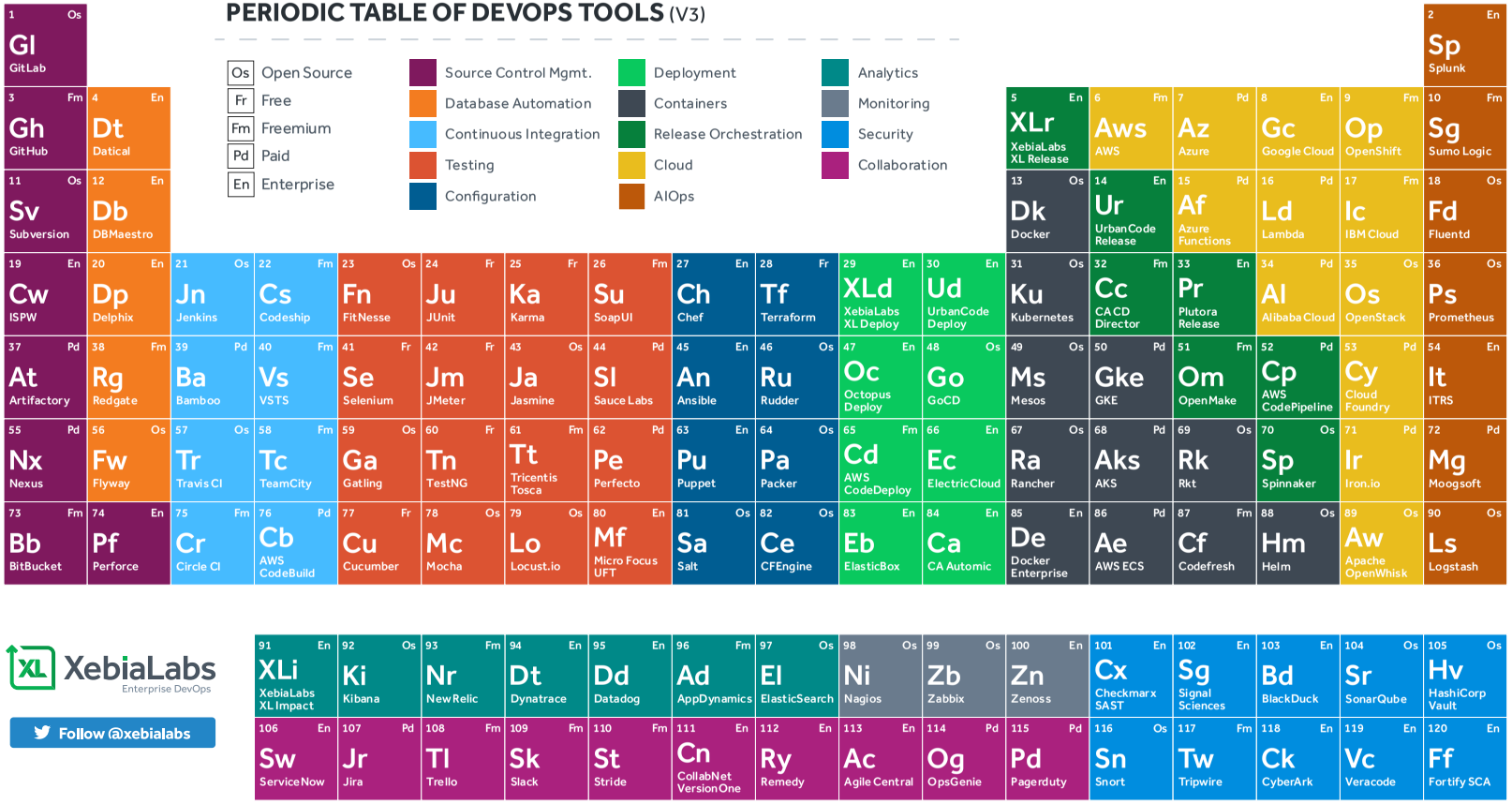


Figure 2.3: The DevOps Periodic Table from Xebialabs.

(2) *a plan or protocol for carrying out or accomplishing something*, and (3) *an underlying scheme that governs functioning, developing, or unfolding* [Mer].

In the context of software engineering, the definition still applies, with the plan being made in software design being the plan of how the software is to be implemented. Ralph and Wand highlighted the need for a formal definition of what design is in the context of software. They have generically described software design as the *a specification of an object, manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints.* In the context of Object-Oriented Programming (OOP), Booch described abstraction, encapsulation, modularization, and hierarchy as the fundamental principles of software design [Boo04].

During the design phase, the engineer needs to evaluate the software's requirements and create a viable plan for its implementation, respecting the known present and future constraints. Faced with this challenge, what resources are there available to support design decisions? In his Ph.D. dissertation, Christopher Alexander claims that most information on any specific body of knowledge is *hard to handle, widespread, diffuse, unorganized and ever-changing*, that this *amount of information is increasingly growing beyond the reach of a single designer* [Ale64], which lead us to understand that the design process is increasingly complex due to the vast amount of incongruous information available. To address this issue, Alexander introduced the concept of design patterns.

2.6.1 Software Design Patterns

Based on the work of Christopher Alexander, *A Pattern Language*, where he presented patterns for architecture and urban design [AIS77], software engineers recognized the relevance of patterns and pattern languages as a strategy for disseminating their design knowledge [Ale02; Ale79; Ale64]. In the context of software design, patterns continue to refer to recurrent problems and their respective solutions, along with the forces that need to be balanced for making the solution fit.

Design Patterns and pattern languages were first introduced in the context of software engineering by Kent Beck and Ward Cunningham at OOPSLA[5] conference in 1987 [Sow87]. They recognized pattern languages for sharing design knowledge, which could significantly improve the selection and application of design abstractions [Sow87].

[5] Object-oriented Programming, Systems, Languages, and Applications, now part of ACM SIGPLAN conference on Systems, Programming, Languages, and Applications: Software for Humanity.

By 1993, the Hillside Group[6] was founded as a non-profit organization with the mission to disseminate the usage of patterns. By 1994, the first substantial contribution to software engineering through patterns was published, with the book *Design Patterns: Elements of Reusable Object-Oriented Software*, detailing a pattern language for object-oriented programming [Gam+94].

Another usage for patterns and their languages was proposed by Christian Kohls, who claims that patterns can be used to describe new theories as part of a scientific process to propose and document new knowledge [KP10], even before its observation in the wild. These are often called *proposed patterns* and follow the same structure, but disregard the description of the known usage section. They can be used between engineers to share potential solutions to their problems, even if not validated in practice.

2.6.2 The Epistemology of Patterns

By definition, patterns are the observation of a recurrent solution to a problem. Authors write their patterns based on observation and experience, with any bias this might contain. Still, engineers trust them as a reliable source of knowledge for guiding their design. Why are engineers trusting pattern authors without negative results? Should they be doing it?

To ensure the validity of their patterns, authors often comply with the *rule of three*, a practice that recommends at least three coherent observations of the solution before documenting it as a pattern [KP10]. These known uses are one of the sections from most pattern structures. However, is this a scientific approach? Would it be possible for an author to observe a skewed reality and capture a misleading pattern? Yes, but it would not be likely to happen. Kohls argues that it is unlikely that an author observes three identical and independent solutions for the same problem, where the identified solution does not fit the problem [KP10]. The most famous book on software design by the Gang of Four (GoF) only described two known uses per pattern [Gam+94].

While the rule of three might ensure pattern with relevant solutions, that does not mean that they are scientifically valid. Following their definition, patterns are, in fact, items of empirically gathered knowledge, which makes them conjectures that, albeit not scientifically proven, have accumulated enough evidence to at least be considered

[6] The Hillside Group is an American non-profit organization that supports the dissemination of patterns for capturing and sharing knowledge. The organization sponsors several pattern related conferences worldwide, namely PLoP, EuroPLoP, ChiliPLoP, KoalaPLoP, VikingPLoP, SugarloafPLoP, or EduPLoP. They have also been responsible for getting the Pattern Languages Of Program Design series of books put together and published. An academic journal is also published as part of Springer's Lecture Notes in Computer Science (LNCS), entitled Transactions on Pattern Languages of Programming.

as promising theories. In this regard, Kohls states that patterns can be a strategy for proposing new design theories, even before use cases can be identified [KP10]

Epistemologically, what is missing is thus proving the theory, beyond mere recurrence. In fact, the pattern context and forces establish the exact premises the designer would agree with, before considering a pattern for selection. Balancing the forces for a pattern ensures its fitness for a problem. The solution is thus an engineering apparatus that, under proper evaluation, constitutes a complete theory towards being what we commonly call a *recurrent, good solution*. Hence, if we are to focus on the scientific validity of each pattern, we should carefully choose (on a case by case basis), what exactly is one to measure in order to provide evidence of fitness.

Towards pattern validity, the observation process might follow empirical strategies — the pattern community named this process as pattern mining [ISM11; Sas+16; Han98]. Kohls [KP10] states that patterns can be scientifically sound if the mining process follows scientific methodologies as well.

In the research presented in Chapter 3 (p. 25), we highlight how many authors have based their pattern writing in their expertise, providing little empirical evidence of their fitness to the real world, possibly rendering their patterns misleading.

2.7 Summary

The World Wide Web, through cloud computing, enabled developers to build applications targeting users on a global scale. Coping with such unprecedented scale disrupted software design, mostly by reaching the limits of vertical hardware scaling, which popularized horizontal scaling through SOA and Microservice architectures.

The Web 2.0 revolution, along with the introduction of cloud computing, motivated developers to demand new practices and architectures to build web applications. Online businesses became more agile at distributing their software, reducing the software release cycle from months to hours. Manual operations became a bottleneck. The DevOps culture pushed teams to own their operations through automation, expanding on the recommendations for autonomous teams from agile methodologies. The relevance of DevOps led to a technological explosion, with hundreds of tools made available, rendering it difficult for engineers to decide their development stack.

The fast-paced cloud market left little time for experimentation, with decisions being critical for the success of software companies. Software design patterns and their languages, which were popular since the introduction of OOP design patterns, could facilitate

this knowledge and help engineers build systems faster. Still, not all patterns provide scientifically valid knowledge, as further demonstrated in the following chapter.

Chapter 3

Designing Software for the Cloud

Cloud computing enabled teams to build applications that reach users on a global scale. The DevOps culture is expanding the concept of autonomous teams inherited from the agile methodologies to also operate their software, through automation. Still, while designing software for the cloud, engineers face a multitude of novel challenges, characteristics of cloud development, that were not observed before. From having to scale their systems horizontally to ensure the system is up and working as expected or recovering in case of failure, these intricacies make cloud development not trivial. This chapter identifies the existing best practices for building software for the cloud. These can guide developers while engineering their cloud software. We conclude, claiming that the existing body of knowledge is limited in both detail and validity.

3.1 Intricacies from Cloud Software Development

Exploiting cloud computing requires domain-specific knowledge that shifts from traditional software development in several fronts, namely software architecture and team organization. The software itself is no longer the single work item delivered by the engineering team; automation, quality assurance, and operations became equally essential [BCS15; Roc13].

Being a fast-paced market, the ability to deliver software fast is key to success, under

the risk of having a larger competitor copying a slower team and gaining their market share[DJG18]. Just as well, reliability is essential to retain the users' trust [Kim+16].

In this section, we present the conclusions from multiple cloud-related publications, identifying the requirements, challenges, and recurring failures of developing software for the cloud.

3.1.1 A Survey over Cloud Adoption

RightScale's inquired 786 IT professionals during January of 2019 [Rig19] for their annual cloud survey. They report a cloud spend growth, with projected spend for 2019 being 24% higher than in 2018. The 13% largest enterprise spenders have a budget of over $12 million a year for public clouds. 50% spend over $1.2 million annually. A total of 94% of enterprises have a cloud presence, using either public, private, or both clouds. The percentage of respondents adopting at least one public and one private cloud in a multi-cloud approach was 69%. Public cloud has an edge attracting practitioners, being a top priority for 31% of the enterprises. Enterprises are running 33% of their workloads on public clouds and 46% on private clouds. Still, public cloud spend is growing at three times the rate of private cloud usage, with workloads slowly being migrated to public clouds.

The report highlights the following challenges with development for the cloud:

Lack of expertise. As stated in Chapter 2 (p. 11), cloud computing promoted a quick growth of a new software paradigm, leading to an explosion of technologies and software designs that support it. This rapid expansion made it very hard for engineers to stay on par with the new trends, requiring considerable research from most to initiate their cloud development. While some became proficient with cloud, they were insufficient to fulfill the demand for engineers savvy in cloud computing.

Cloud security. Data breaches or misuse has been a frequent discussion, as cloud products exponentially increase the volume of data they capture from their users. Data privacy laws, such as the European General Data Protection Regulation (GDPR) [Uni16], along with security concerns, can constrain development teams from trivially moving their applications to private clouds, but are a necessity for ensuring data protection and a breach that could degrade the customers trust in the product. Puthal claims that customers need to trust their cloud providers. Each cloud layer addresses its security, and every layer must trust its underlying layers to be trustful itself [Put+15]. As such, security threats resulting from adopting

third-party services are lower in Infrastructure as a Service (IaaS), increasing with Platform as a Service (PaaS) and highest with Software as a Service (SaaS) since service complexity increases at each of those levels. Private clouds tend to be more secure than public clouds because these exist in dedicated hardware that operates in isolated computer networks. Limiting the exposure to the Internet and not sharing resources between multiple entities results in a reduction of the attack vectors to it [RES10]. Data privacy in the cloud is another primary concern. Some cloud providers lack external auditing that ensures their security practices [MK10]. Despite that, cloud adopters will have to trust their cloud providers with their private data.

Governance. Cloud governance relates to the process of cloud operation. It answers the questions of how cloud orchestration should operate, addressing accountability, decision rights, risk, and resource management [Gro16]. It addresses specific questions such as change permissions, balancing forces like agility versus risk.

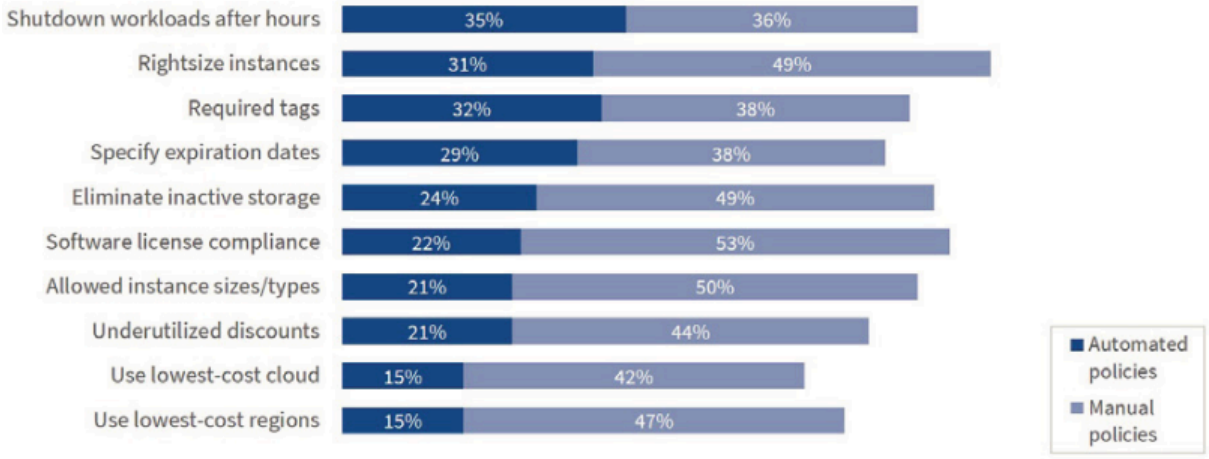


Figure 3.1: Cloud budget optimization initiatives. Shutting down after-hours workloads, adjusting instance sizes and labeling resources were the most relevant automated policies [Rig19].

Cloud spend. Cloud computing facilitates access to computing resources at scale, but it still introduces a relevant impact on the monthly budget for companies developing software for the cloud. Over 50% of the respondents claim to spend over $1.2 million in it. There are several optimization strategies, such as yearly contracts or resource optimization, which are not trivial for an engineer unfamiliar with them. Figure 3.1 (p. 27) lists the most adopted cost optimization strategies, according to the surveys' respondents. Considering both manual and automated initiatives, adjusting instance sizes is the most frequently applied strategy for cost optimization.

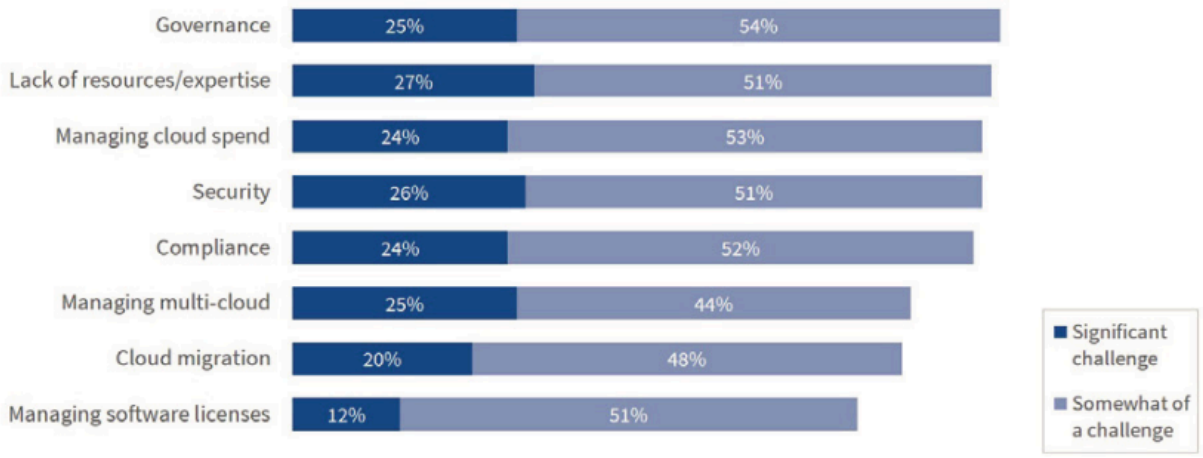


Figure 3.2: Cloud challenges for 2019. Overall, the lack of resources and expertise are the most important open challenges in adopting cloud computing. Credits: RightScale cloud report [Rig19].

BEGINNER	INTERMEDIATE	ADVANCED
1. Governance (86%)	**1. Governance (78%)**	**1. Governance (78%)**
2. Lack of resources/expertise (85%)	**2. Managing cloud spend (77%)**	**2. Managing cloud spend (76%)**
3. Managing cloud spend (84%)	3. Lack of resources/expertise (76%)	3. Compliance (76%)
4. Cloud migration (83%)	4. Security (76%)	4. Security (74%)
5. Security (82%)	5. Compliance (73%)	5. Lack of resources/expertise (73%)

Figure 3.3: Cloud challenges for 2019 by business maturity. Governance is a shared concern across all respondents, with cloud spend and lack of resources or expertise following closely. Credits: RightScale cloud report [Rig19].

We can see how vital these challenges are for the respondents in Figure 3.2 (p. 28) and broken down by the level of expertise from the respondents in Figure 3.3 (p. 28). Still, in 2019, the most significant challenge for cloud adoption is the lack of resources and knowledge available to guide engineers, closely followed by governance and multi-cloud management.

3.1.2 Concerns from Cloud Design

Designing software for the cloud imposes a complex paradigm shift from traditional software development [UX13]. Developers now have to consider requirements that were non-existent in the past, namely:

Scalable by design. When designing scalable software, it is expected that, when an application meets its vertical scalability limit, it can continue to scale horizontally [Ace+13; Sti15]. By design, monolithic services can only scale vertically, which deems them unfit for scalable cloud architectures [Bon16]. New architectures, such as Microservices, must be adopted to separate services into smaller components, which scale individually [VW12; UX13], facilitating coping with the variable demand often seen in cloud application [DL12; Put+15]. Scalability must concern all application components, such as data storage, messaging infrastructure, and more [Ric17b]. Nevertheless, Fowler raises awareness of the risks of preemptive optimization, stating that it is the natural evolution for services to start as monoliths and to be broken into microservices when needed [Fow15].

Dynamic infrastructure. To take advantage of microservice architectures and enable cost-efficient scalability, the infrastructure needs to adapt to the application's load, scaling itself up and down dynamically [VW12; DL12; Sav11; Put+15; Ace+13].

Service orchestration. While working at a large scale with services that exist across tens to hundreds of servers, it is not practical to manage services running in each server individually. Servers' status and resource usage need to be abstracted, enabling service placement in the cluster, with its allocation to a specific server being autonomous. Just as well, the cluster should be autonomous at identifying failing service instances, performing the required actions for recovering it: either restart it or start it in a different server [DL12; Rig17].

Service discovery. Deploying services in dynamic infrastructure introduces the need for them to discover each other so that they can cooperate as expected. Dynamically infrastructure makes this harder, as there is no fixed service address at the time of deployment [Ace+13; TCB14].

Monitoring. Software fails. When deploying to the cloud, the software will run on dynamically provisioned hardware, possibly in many geographically distributed servers. To ensure that the system is running as expected, automated monitoring of every server and service is essential to enable teams to detect and fix issues faster [UX13; DL12; Put+15; Ace+13; Mic19].

Software isolation. Cloud computing provides computational resources using shared servers and virtual machines. It is often observed the need to deploy multiple services to the same virtual machine. When deploying software, all its dependencies

must be made available. Changing the virtual machine to accommodate a specific service's dependencies will leave a trail of files when that service is removed [Fel+12]. Additionally, deploying multiple services in the same virtual machine might result in dependencies incompatibilities [Hwa+13]. Proper software isolation is required to ensure that the service's binary files and all of its dependencies are deployed, providing an optimal environment for running the service. The same isolation should ensure that the service can be upgraded and removed in the future without polluting the virtual machine with its specific configurations or dependencies [UX13; Rig17].

Messaging. The distributed nature of cloud applications requires a messaging infrastructure that facilitates communication between services in a loosely coupled manner to enable scalability [Cas+11]. Asynchronous messaging is widely used and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more [Mic19; Gar06].

Availability. Availability, or the percentage of time the system is available, is essential for complying with contracts and keeping clients and users happy. Disruptions can happen due to software and hardware errors, malicious attacks, and unmanaged system load. Cloud applications typically provide users with a Service-level Agreement (SLA), so applications must be designed to ensure their availability [RES10; BHS07; UX13; Ace+13; Mic19];

Reliability. Reliabilities measures the probability of a system to produce the expected output throughout time [UX13]. A reliable service performs consistently according to what it was designed to do [Ace+13].

Resiliency. Resiliency evaluates the ability of a system to handle and recover from failures gracefully, often referred to as being fault-tolerant [Mic19]. Cloud applications often use shared platform services, are multi-tenant, compete for resources and bandwidth, communicate over the Internet, and run on commodity hardware means there is an increased likelihood that both transient and more permanent faults will arise [BP19]. Detecting failures, and recovering quickly and efficiently, is necessary to maintain resiliency [ECN15; Bir15].

Security. Exposed to the internet, cloud services are ideal targets for hacker attacks. The software must be designed with security considerations to minimize the probability of being compromised, while infrastructure must be carefully protected to prevent unauthorized access to resources and data [RES10; Mic19; Gre+08; Arm+10; Put+15].

Complying with these requirements provides the foundation for adequately designing software for the cloud. For inexperienced engineers, these will require a sheer volume of investment in Research and Development (R&D), with bad decisions being the most likely reason for generating failures.

3.1.3 Cloud Failures

Cloud software, like all software, is subject to failure. We have discussed how reliability influences user retention, which implies that failures can be the genesis for the termination of some cloud businesses. The following failures have been identified as recurrent by Gadish [Gad14]:

Human error. The introduction of human error in software is not a matter of if it will happen, but when it will happen. From development to operations, each day without human error increases the likelihood that an error will be introduced soon. Hollnagel describes human error as an identifiable human action that is seen as being the cause of an unwanted outcome. He evaluates human reliability assessment strategies, namely how those could be predicted [Hol05]. In the context of the cloud, human error has a potentially catastrophic impact. Puthal describes how mistakes can lead to the loss of cloud storage [Put+15]. Edelman describes how infrastructure changes can be erroneous and interrupt service provisioning [Ede]. Greenberg describes how humans can introduce significant performance issues [Gre+08]. Also, a debugging exercise in 2017 at Amazon Web Services (AWS) resulted in service downtime and the estimated loss of $150 million for major American companies [Sve17].

Application bugs. Application failure due to software bugs is another recurring cause of failure. Traditional IT practices can help mitigate the issue, through automation pipelines for testing and deploying applications [EAD14; Roc13; Día+18; Sha15; Tec14]. Armbrust describes how bugs in cloud applications are hard to debug, given that it is often complex to expose the software to the level of stress it reaches in production [Arm+10].

Cloud provider downtime. Cloud providers are becoming very reliable, providing the tools and strategies to ensure redundancy for increased reliability. For example, AWS provides multiple availability zones in each deployment region so that applications can be deployed redundantly in a specific geography [Ser19b]. Still, some issues are impossible to prevent, such as natural causes, and even top-tier providers might be affected. A minor network disruption in AWS in 2015 led to over five hours of

intermittent service in multiple services, critical for many clients, rendering their services unavailable [Ser15]. An outage in AWS during the 28th of February 2017 started by taking down Simple Storage Service (S3) and quickly degraded other services as well. AWS status dashboard itself was malfunctioning, as it depended on S3. The cause was eventually attributed to human error [Ser17]. Multiple cloud applications were affected, namely Medium or Slack.

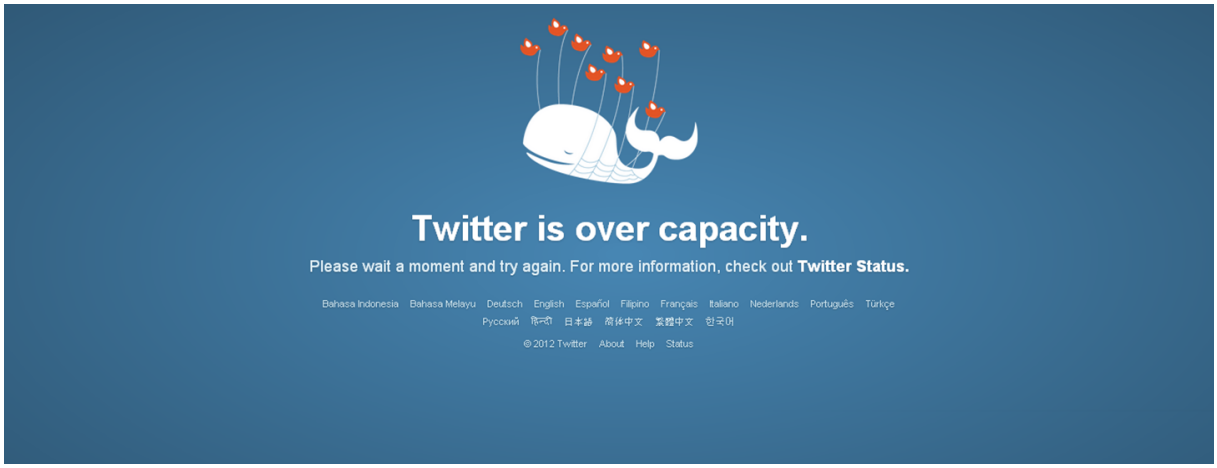


Figure 3.4: The Fail Whale from Twitter, a static page often seen by the social network's users during its first years due to inability to accommodate user demand.

Quality of service. The definition of a functioning service varies depending on the business provided. While it might be acceptable for some applications to have increased response times during some part of the day, to others, such as the video or music streaming industries, any unexpected latency or inability to stream content has a critical impact on the business. Also, user demand can harm the infrastructure if resources are not preemptively allocated configured to scale automatically. A well-known story of service disruption was observable during the early years of Twitter, a time during which it was common to see their Fail Whale, depicted in Figure 3.4 (p. 32), a static page that reported that something was not working as expected with the application.

Privacy and security breaches. Security requirements are essential to ensure the privacy of customer data and, into some businesses, mandatory for operation, such as the GDPR [Uni16]. Enterprise clients often request for information security management certifications before adjudicating their business, such as the ISO/IEC 27000 family of certifications [Int]. Cloud providers themselves will not be able to ensure that deployed applications are accurate, and, as such, it is still up to the development teams to ensure the continued security of their application.

Lack of disaster recovery procedures. Disaster recovery has been common practice for decades in physical data centers [Wol06]. Cloud providers offer a multitude of strategies to address failure: multi-zone deployments, automated backups, automated recovery on failure, and much more. These can lead to relaxation by the engineering team, delegating the responsibility of business continuity to their cloud provider. Such relaxation has more than once led the team to believe that they had proper recovery strategies in place, only to learn that they were not properly configured when they were needed. GitLab's 2017 backup wipe and recovery description is an excellent example of this failure: the team believed they had multiple backup strategies in place, but most were not working as expected. After a human error led to a partial database wipe, they took several hours to find a viable recovery solution and ended up having to recover the database from cold storage, which took over 18 hours [Git17].

3.1.4 Discussion

Developing software for the cloud extends traditional software development with a multitude of requirements that demand extensive learning and practice from newcomer engineers. The explosive growth in cloud technologies introduced hundreds of new tools, services, and architectures for addressing cloud problems [Xeb19b], which further increases the entropy for finding optimal solutions. Designing cloud software requires months of research and experimentation to reach an ideal balance of cloud design and technologies. Adding experts to the team can mitigate the time required to reach productivity, but experts might not be available, either due to their increased demand in the market or due to limited human resources budget [Rig19].

Failure to make the best design decisions can lead to extensive damages as failures appear. We argue that cloud design decisions are critical to ensure that cloud requirements are addressed while mitigating the likelihood of failure. It is paramount that engineers have resources at their disposal, which can support their decisions to build and operate cloud software better.

3.2 Cloud Design Patterns

Section 2.6 (p. 19) describes how patterns are a strategy for sharing knowledge regarding recurring design problems and their solutions. The relevance of patterns amongst Software Engineering communities led cloud computing researchers to also adopted

them to document their observed problems and solutions. Design patterns can help development teams bootstrap their cloud development with confidence, reaching maturity faster [Feh+14]. This section describes notable usages of design patterns for describing architectures for creating cloud computing infrastructures and applications.

3.2.1 Arcitura Cloud Patterns

Arcitura Education[1] is a global provider of progressive, vendor-neutral training and certification programs. Their programs are focused on Information Technology (IT), big data, cloud, and Service Oriented Architecture (SOA). As part of their teaching strategy, they develop their teaching materials, including multiple pattern books for the cloud, namely *Cloud Computing Design Patterns* and *Cloud Computing: Concepts, Technology & Architecture* [ECN15; EMP19]. The books present a catalog of patterns for cloud computing, covering a vast array of topics, namely scaling, resilience, monitoring, data management, storage, or containerization. An overview of these patterns is freely available on their site [Arc19].

The authors describe at the time 89 cloud-related patterns in their work. More patterns are available for SOA, microservices and containerization, DevOps, and other topics. While extensive in numbers, each pattern includes only a single sentence for the problem, solution, and application, some accompanied by a visual representation. It enables the reader to get acquainted with the problem and the concept of its solution. It lacks proper identification and discussion of the forces for a deeper understanding of the problem and the description of known uses for the patterns.

3.2.2 Cloud Computing Patterns Book

The book Cloud Computing Patterns [Feh+14] describes cloud components and practices. A map of the patterns and their relations is depicted in Figure 3.5 (p. 35). Patterns are organized in the following categories:

Cloud computing fundamentals. Describes cloud service models and deployment strategies, elaborating on how to decide between them considering for different applications.

Cloud offerings. Describes the services available through cloud providers and how to leverage them for building applications.

[1] Learn more at https://www.arcitura.com/about/.

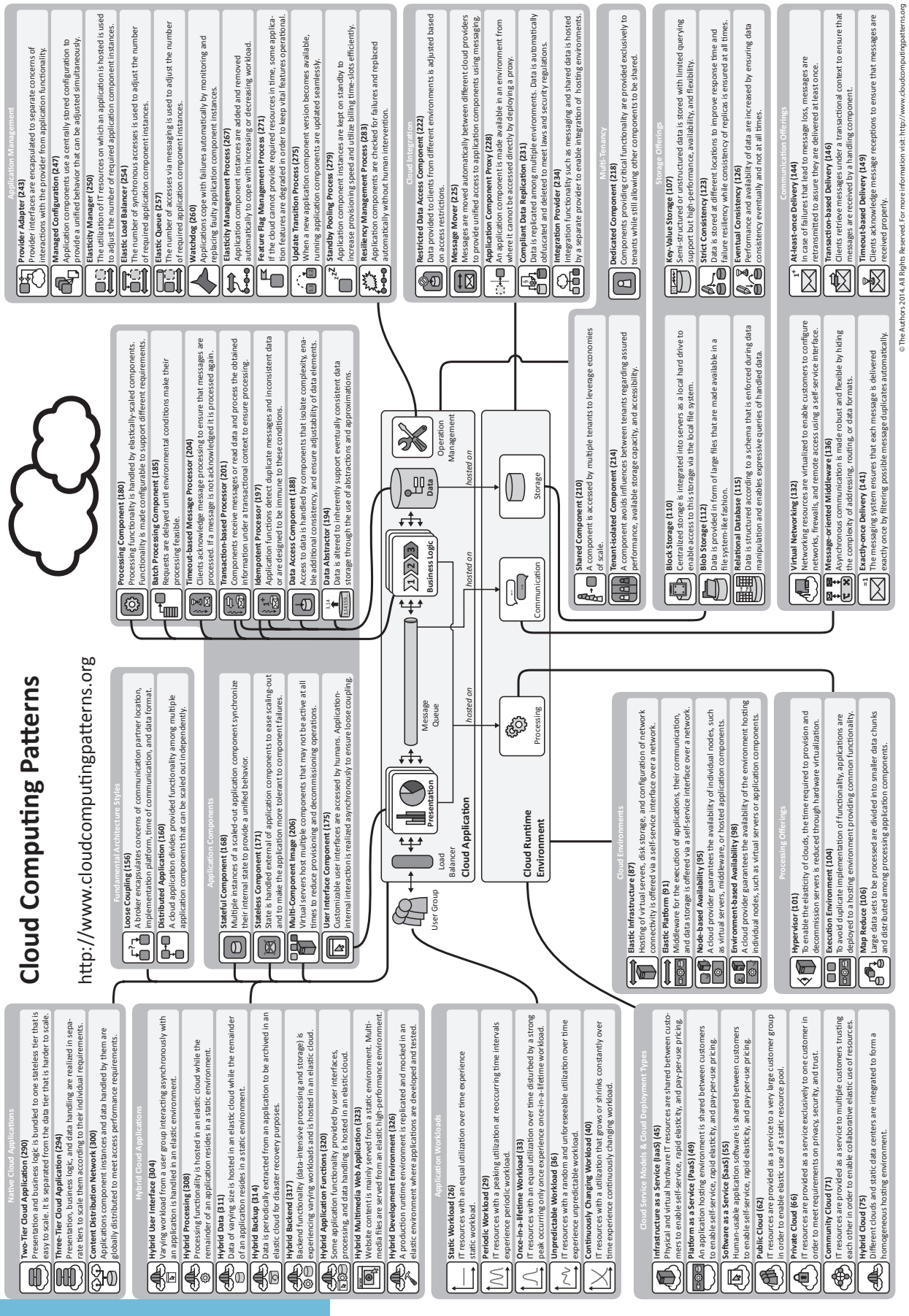


Figure 3.5: A visual representation of the cloud computing Patterns.

Cloud application architectures. Details how applications should be organized in the cloud.

Cloud application management. Elaborates on how applications should be managed to ensure they continuously work as expected.

Composite cloud applications. Leverages multiple patterns together to address complex scenarios.

These patterns are composed of an abstract to the pattern, the problem in the form of a question, the context, solution, result, solution variation when applicable, related patterns, and known uses. A visual representation often supports the solution.

The authors did not identify and discuss the forces for each pattern, and the description of the known uses use generic examples, lacking factual application scenarios of the pattern. The book is still thorough on covering many details of cloud computing, resulting in an excellent reference over the properties of cloud development.

3.2.3 Amazon Web Services Reference Architectures

AWS Reference Architectures [Ama] is a repository of reference architectures by Amazon, describing suggested solutions to specific application scenarios such as web application hosting or batch processing. Amazon currently provides a total of 16 reference architectures. Each reference architectures is visually described, identifying the services that compose the solution and how they interact with each other. These guidelines are not written in the form of patterns and are agnostic of the context where the solution can be implemented.

Figure 3.6 (p. 37) depicts AWS web hosting reference architecture. Each service composing the solution is briefly described on the bottom of the image, enabling the reader to understand its relevance in the bigger picture.

3.2.4 Azure Design Patterns

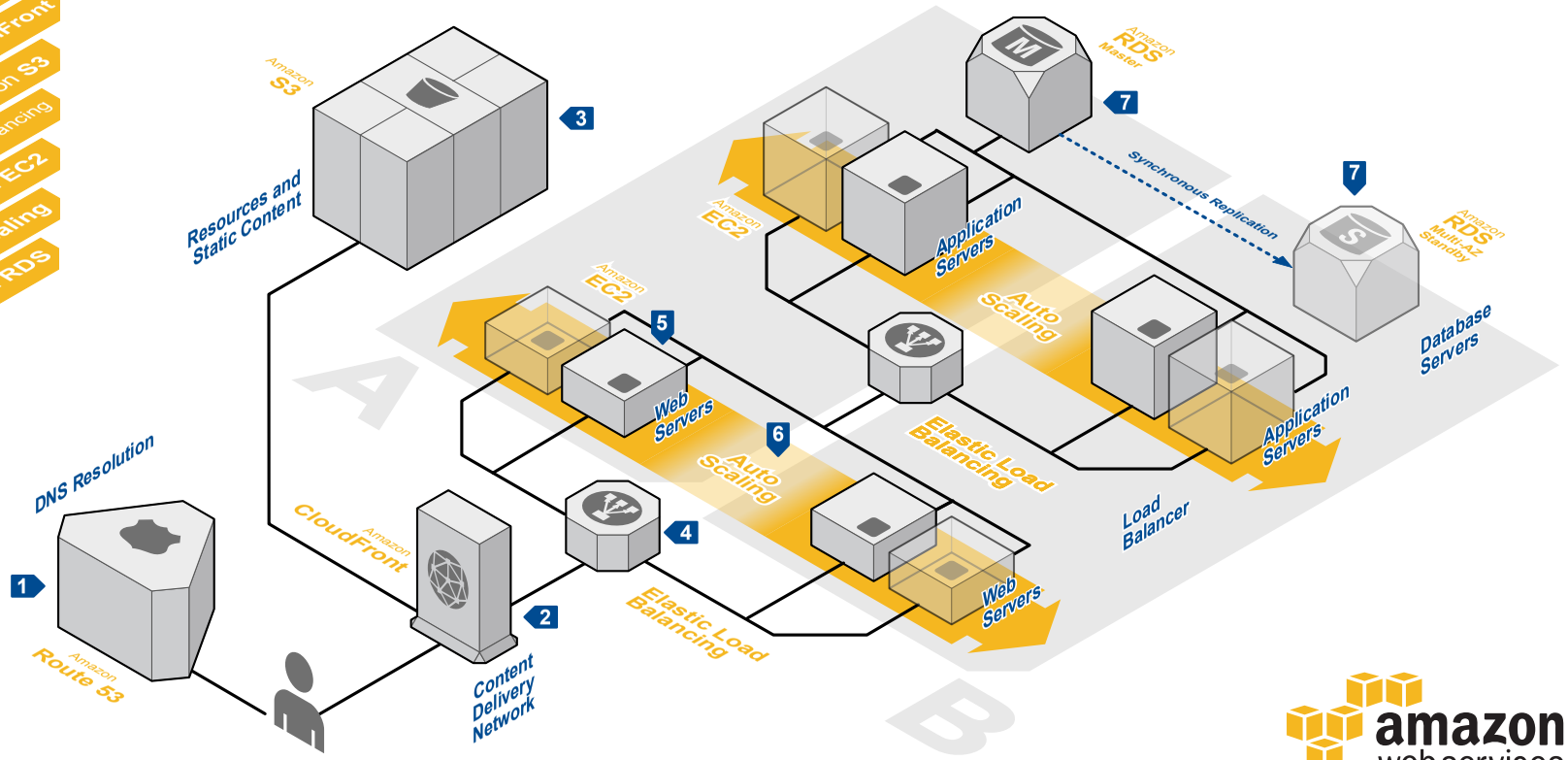
Azure provides at this time a list of 24 design patterns for cloud computing. Azure design patterns follow a traditional pattern structure, including a context and problem description, the solution, a description of issues and considerations while adopting the pattern, an example, either as code or described using diagrams, and a list of related patterns. In some cases, a downloadable example is available [Mic19]. The patterns' abstracts are listed in Table 3.1 (p. 38) and Table 3.2 (p. 39).

AWS Reference Architectures

Amazon Route 53
Amazon CloudFront
Amazon S3
Elastic Load Balancing
Amazon EC2
Auto Scaling
Amazon RDS

WEB APPLICATION HOSTING

Highly available and scalable web hosting can be complex and expensive. Dense peak periods and wild swings in traffic patterns result in low utilization of expensive hardware. Amazon Web Services provides the reliable, scalable, secure, and high-performance infrastructure required for web applications while enabling an elastic, scale-out and scale-down infrastructure to match IT costs in real time as customer traffic fluctuates.



amazon web services

System Overview

1 The user's DNS requests are served by **Amazon Route 53**, a highly available Domain Name System (DNS) service. Network traffic is routed to infrastructure running in Amazon Web Services.

2 Static, streaming, and dynamic content is delivered by **Amazon CloudFront**, a global network of edge locations. Requests are automatically routed to the nearest edge location, so content is delivered with the best possible performance.

3 Resources and static content used by the web application are stored on **Amazon Simple Storage Service (S3)**, a highly durable storage infrastructure designed for mission-critical and primary data storage.

4 HTTP requests are first handled by **Elastic Load Balancing**, which automatically distributes incoming application traffic among multiple **Amazon Elastic Compute Cloud (EC2)** instances across Availability Zones (AZs). It enables even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic.

5 Web servers and application servers are deployed on Amazon EC2 instances. Most organizations will select an **Amazon Machine Image (AMI)** and then customize it to their needs. This custom AMI will then become the starting point for future web development.

6 Web servers and application servers are deployed in an **Auto Scaling** group. Auto Scaling automatically adjusts your capacity up or down according to conditions you define. With Auto Scaling, you can ensure that the number of **Amazon EC2** instances you're using increases seamlessly during demand spikes to maintain performance and decreases automatically during demand to minimize costs.

7 To provide high availability, the relational database that contains application's data is hosted redundantly on a multi-AZ (multiple Availability Zones–zones A and B here) deployment of **Amazon Relational Database Service** (Amazon RDS).

Figure 3.6: Amazon Web Services web hosting reference architecture.

Pattern Name	Description
Cache-Aside	Load data on demand into a cache from a data store
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource
Command and Query Responsibility Segregation	Segregate operations that read data from operations that update data by using separate interfaces
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain
External Configuration Store	Move configuration information out of the application deployment package to a centralized location
Federated Identity	Delegate authentication to an external identity provider
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service validates and sanitizes requests, and passes requests and data between them
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries

Table 3.1: Summary for the first twelve Azure design patterns [Mic19].

Pattern Name	Description
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances
Materialized View	Generate prepopulated views over the data in one or more data stores when the data is not ideally formatted for required query operations
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes to smooth intermittent heavy loads
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed
Runtime Reconfiguration	Design an application so that it can be reconfigured without requiring redeployment or restarting the application
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources
Sharding	Divide a data store into a set of horizontal partitions or shards
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client
Throttling	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Table 3.2: Summary for the last twelve Azure design patterns [Mic19].

3.2.5 Pattern Language for Microservices

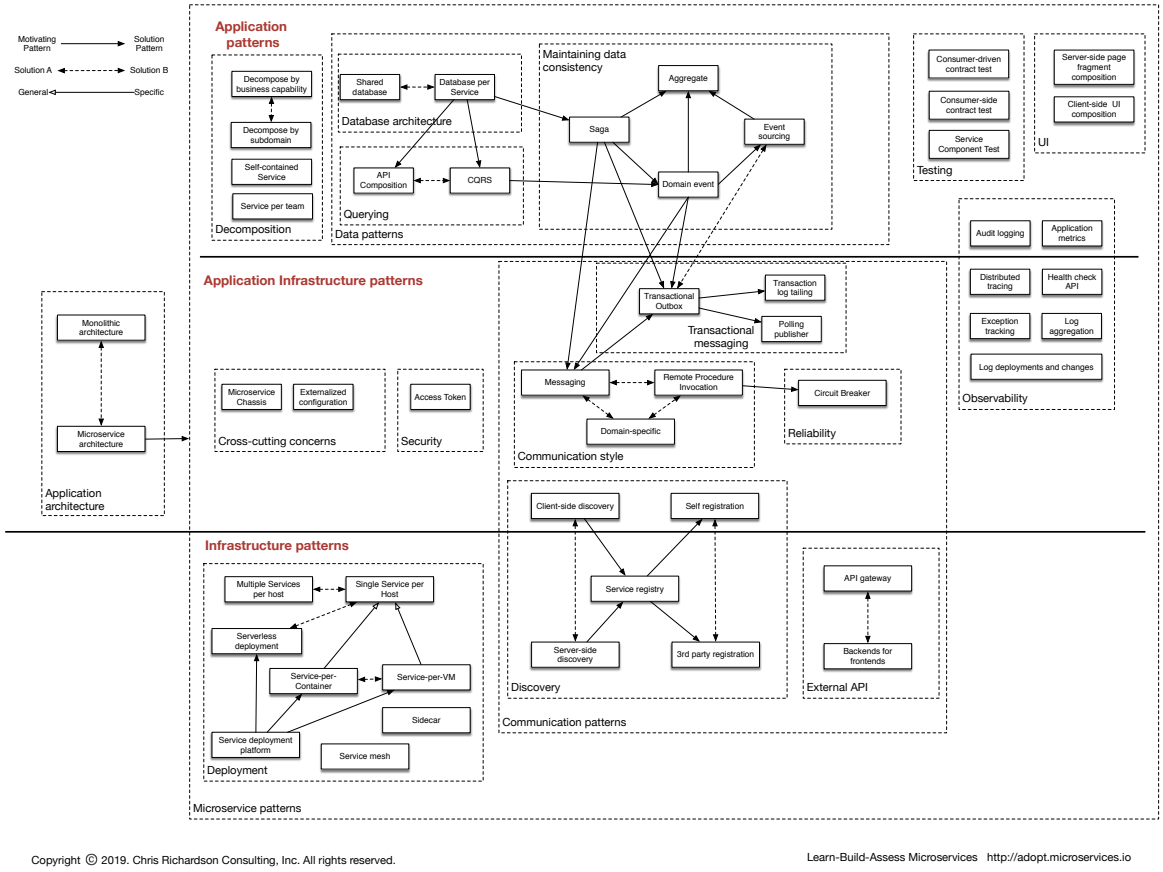


Figure 3.7: Chris Richardson pattern map representing his pattern language for managing microservices.

Chris Richardson wrote the *Microservices Patterns* book [Ric17b], along with his website[2], where he describes a pattern language for building and orchestrating microservices in the cloud [Ric17a]. So far, the pattern language is composed of more than 40 patterns, depicted in Figure 3.7 (p. 40). The language is being continuously expanded, from the knowledge gathered by Richardson consulting services regarding microservices architectures. While both the book and the web site approach the same topic, the website has a description of the patterns in pattern form, using a problem, forces, solution, example, resulting context, and related patterns form. The book does not describe the patterns in this format but, instead, demonstrates how they can be implemented technically, including several source code examples and architecture diagrams.

[2] Available at https://microservices.io/patterns/

3.2.6 Delivery Patterns

Cycligent engineering team published a white paper of cloud patterns for Continuous Integration (CI) and Continuous Deployment (CD). The patterns are described informally, providing a brief description of the solution and known use cases for them [Cyc15]. The patterns described are:

Blue/Green deployment. A deployment strategy where updates are made available for a subset of the users to test changes minimizing possible negative impacts. The current environment is typically labeled as *blue*, while the updated one is the *green*. A routing strategy forwards the users from the blue to the green environment. In case of failures, users can be reverted to the blue environment. Netflix, Localytics, and Etsy are described as adopters of this strategy [Cyc15].

Canary release. Very similar to Blue/Green deployments, but enabling a gradual migration of the users, instead of the all or nothing approach from blue/green deployments. If the canary version is stable, additional users can be gradually switched to this environment, until no users are left, when the previous environment is shut down. Facebook, Box, and Wix are known to adopt this strategy [Cyc15].

Microservices. Already extensively described in the literature, Cycligent also highlights the relevance of microservices for cloud applications and claim that Nike, Karma, and Netflix are adopting them [Cyc15].

Containers. Also extensively described in the literature, Cycligent highlights the importance of immutable environments built with containers for facilitated isolation and portability.

Dark launching. Introduced in 2008 by Facebook [Cyc15], it consists of pushing hidden features to the users and evaluate how the application behaves. If no abnormal behavior is observed, the feature can gradually or at once be made available.

Feature flags. It can be used along with Dark Launching to identify to which users new features should be made available. It facilitates toggling features for specific users based on configuration or rules, for example, on the user's details such as the country of origin. Hootsuit, Spotify, and Flickr are said to adopt this strategy [Cyc15].

These set of practices are only briefly described and do not follow a pattern structure but are relevant to the reader to get acquainted with CI/CD strategies and the companies adopting them.

3.2.7 Other Works

Duvall summarized CD patterns and anti-patterns in the software life cycle. He introduces patterns for configuration management, CI, testing, deployment pipeline, builds and deployment scripting, deploying and releasing applications, infrastructure and environments, data, incremental development, collaboration, and references some essential tools [Duv10]. For each topic, there is a single paragraph describing the pattern, with a related anti-pattern. For example, for *parallel tests*, the pattern is *Run multiple tests in parallel across hardware instances to decrease the time in running tests* and the anti-pattern *Running tests on one machine or instance. Running dependent tests that cannot be run in parallel.* Duvall describes fifty patterns with this strategy, which can raise the developer's awareness for these solutions, and how to prevent implementation errors with the anti-patterns description. Still, these summaries are insufficient to detail how the developer could pursue an implementation.

Tsai describes how SOA architectures can be extended with the infrastructure provided by cloud computing to make larger, more complex application [TSB10]. Tsai describes how multiple clouds can be leveraged to increase further redundancy and, hence, the cloud application's availability. The work describes the need for multiple ontologies to be defined and adopted by cloud providers to describe their cloud services, namely regarding storage, computing, and communication, facilitating service allocation across cloud providers. He moves on to theorize the concept of cloud brokers, that would be able to allocate resources from multiple clouds described by such ontologies, facilitating cloud portability and interoperability. A reference architecture of a multi-cloud application is described and demonstrated in a prototype, which used Google Cloud for computation and Microsoft Azure for a SQL database. This concept had mild adoption by cloud providers. As an example, AWS provides Service Broker, which allows AWS services to be exposed directly through third-party applications such as Red Hat OpenShift. This level of maturity is only pursued by extensive cooperation and not really under the domain of our research.

Zimmermann et al. describe a pattern language for creating and evolving microservices. While not directly related to cloud computing, their pattern language provides valuable knowledge for designing services that scale [Zim+].

Friedrichsen introduces 22 patterns in his presentation Patterns of resilience [Fri14]. The patterns are informally and briefly described along a slide deck, and a pattern map is facilitated to navigate the language, as depicted in Figure 3.8 (p. 43). The language is particularly relevant to provide a vocabulary for discussing resilience, as the patterns are insufficiently described to be applied by a development team.

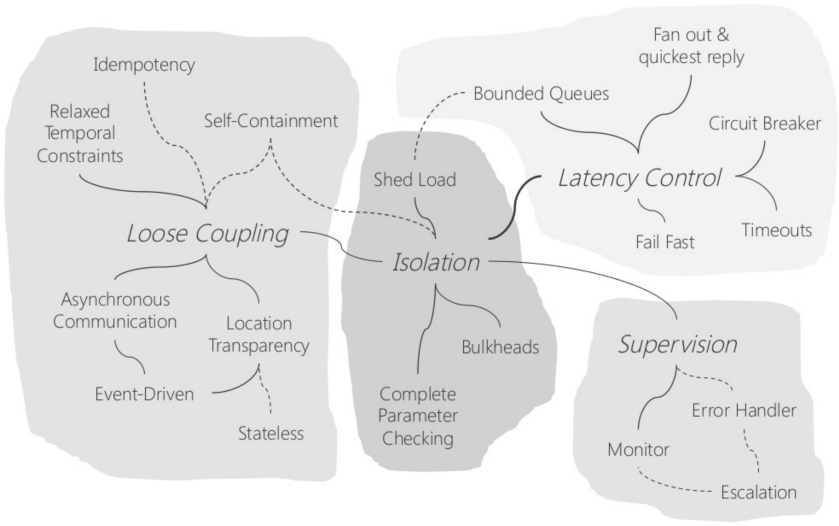


Figure 3.8: Friedrichsen pattern map representing his pattern language for resilience in cloud applications.

Loriedo created a pattern catalog specific for container usage. The patterns are organized into three categories: *development*, with 6 patterns, 2 patterns in *distribution* and 10 patterns for *runtime*. They are made available online in the form of a presentation. Each pattern briefly describes a container-related context, with a brief description and diagram demonstrating its usage [Lor19]. The reader learns about relevant container concepts such as CONTAINER LAUNCHER or MOUNT SOURCES.

The Cloud Native Landscape is an initiative from the Cloud Native Computing Foundation that aims to identify and categorize the most relevant tools for supporting cloud development. They have aggregated a total of 1171 tools to date, categorized under application definition and development, orchestration and management, runtime, provisioning, platforms, observability, and analysis, serverless, and special (for others) [Fou19]. The ecosystem is depicted in Figure 3.10 (p. 45).

3.2.8 Discussion

Cloud design patterns are a recognized asset for empowering engineers to build better software. That motivated authors to publish their design knowledge as patterns, as identified in this section. Despite the availability of such knowledge, one of the main challenges of cloud development is still the lack of resources and experienced engineers for building cloud applications, cf. Section 3.1.1 (p. 26).

From this literature review, we understand that the knowledge being made available is vast. However, it rarely provides concrete implementation details. When it does, it is

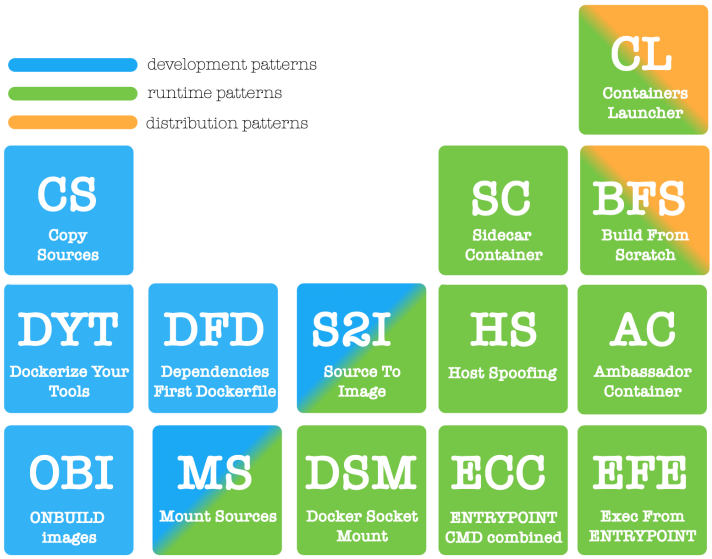


Figure 3.9: Visual representation of the container catalog of patterns from Loriedo, organized into three color-coded categories: development, runtime, and distribution.

often too strict and not adaptable to different contexts. Furthermore, authors tend to document knowledge from their own experience without verifying its validity with other professionals. There has been no empirical study evaluating the impact of patterns as guidelines for defining the architecture of cloud application to the best of our knowledge. No author presents a case study with practitioners evaluating the correctness and relevance of their patterns, which let us believe that these are a product of personal experience and not scientifically tested to ensure their relevance.

We believe that patterns are a good strategy for capturing design knowledge for the cloud, and a pattern language for the cloud is ideal for supporting design decisions. Such patterns would have to go further than the current state of the art, leveraging forces and alternative solutions, to be applicable by most professionals.

3.3 Summary

Cloud computing demanded a considerable shift from software engineers to adapt to the newly introduced requirements. RightScale's survey for 2019 shows that lack of resources or expertise is the most relevant cloud adoption challenge [Rig19]. This is a factor in the introduction of new software requirements such as scalability by design, dynamic infrastructure, service orchestration, availability, reliability, resilience, and others. For a multitude of reasons, cloud failures are an ordinary reality among practitioners, due to human error, bugs, provider downtime, security, lack of recovery procedures, and others.

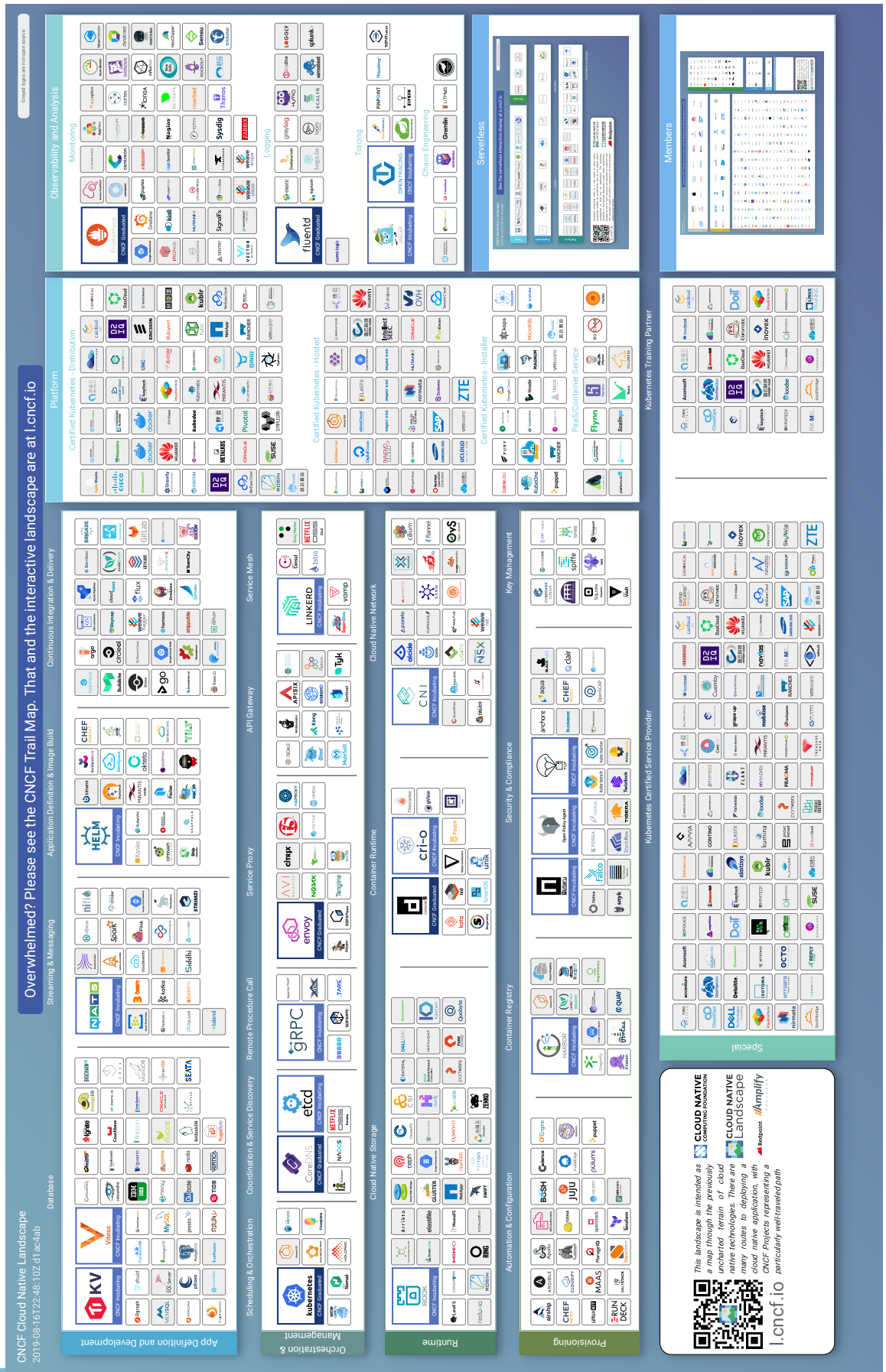


Figure 3.10: Cloud tool ecosystem, according to the Cloud Native Computing Foundation.

Good software design knowledge and practices could improve this scenario, and patterns provide the ideal way of sharing that knowledge.

Cloud Design patterns can help development teams design architectures that cope with cloud requirements faster, reducing the research time required. We have presented the work from other authors who have produced hundreds of design patterns during this chapter. While these should be improving the cloud success scenario, we have seen that lack of cloud knowledge and expertise is still a dominant issue amongst practitioners.

Chapter 4

Problem Statement

Chapter 3 (p. 25) presents the current challenges of cloud computing and how multiple authors have addressed them using patterns and pattern languages. A multitude of design patterns, tools, and applications are described. Despite those, businesses still struggle to build reliable cloud architectures [Gad14]. This chapter elaborates on how this research contributes to the body of knowledge of cloud software development. We argue that there is a lack of scientific literature describing how to build cloud software, supported by systematically captured knowledge that can help teams build their cloud solutions. We describe the hypothesis to be addressed, the associated research questions, and the research strategy used to assess it.

4.1 Thesis Statement

The information made available for supporting cloud application design is often a result of personal experience and observation, biased to specific application contexts, lacking in scientific support and adaptability. As the central hypothesis from this research, we set ourselves to provide evidence that

> *While engineering software for the cloud, there are categories of recurring problems, which solutions converge from* good design *principles, that adjust*

to the context where they emerge. Their adoption is a consequence of (1) the awareness a team has of a problem, (2) the characteristics of the product and the company, and (3) the way these solutions relate amongst themselves.

These recurrent problems can be grouped in a limited number of categories. We believe that engineers tend to converge into similar solutions by applying sound design principles familiar to them. While doing so, even independently, they tend to converge to a similar solution for the same problem, while adapted to the specific context they observe. For these recurrent solutions to emerge, the team must be aware of the problem, and the problem must be relevant enough to justify investing in it. While there is no golden rule or set of strategies to create optimal cloud applications, engineers are likely to engineer software that will comply better with their cloud requirements when provided with proper resources.

We propose to research these practices, mined through observation and literature review, using a pattern language as a systematization framework for knowledge that can be readily applied by cloud developers. We provide evidence of their relevance in resorting to appropriate research methods that involve practitioners that adopted cloud computing to try to understand how factors such as company and product characteristics might influence not only the adoption of each pattern but also the language as a whole.

4.2 Research Questions

As new engineering paradigms emerge, so do new software approaches. Cloud Computing is no exception. To enable our contribution to the field of cloud engineering, we set ourselves to understand what design decisions influence software development and operation under this paradigm. In this regard, we identify five Research Questions (RQs) for guiding this research.

RQ1. What are the recurrent problems when developing software for the cloud?

What recurrent problems can we identify with the current approaches to cloud development? What consequences result from sub-optimal design decisions?

RQ2. What strategies are adopted for addressing cloud problems?

The cloud problems identified in RQ1 are likely to be observed at some stage by any companies developing for the cloud. How should developers address those problems? Have developers converged to sound design decisions when solving these problems?

If so, is it possible to formalize these design decisions, making them reusable by others? What relations can we identify that could motivate the joint implementation of multiple strategies? What lessons can we learn from successful cloud designs?

RQ3. What driving forces influence how strategies are implemented?

Engineers often face difficulties while building cloud software. Not only is it hard to find the ideal solution for a problem, but the same problem might also suffer from varying configurations, or *forces*, which need to be balanced in a specific way to fit each concrete observation of the problem. As such, engineers would be able to better address cloud problems if they understood the forces involved, enabling a proper adjustment of the solution to fit the problem's specific configuration. In this regard, for the identified problems, what forces influence the configuration of the problem? How can those forces be balanced to make a fit solution for a given context?

RQ4. Are companies that develop software for the cloud aware of these problems and adopt the identified solution?

Previous questions search for cloud problems and their solutions and attempt to document them so that professionals can apply them. Nevertheless, are they relevant for professionals? Do they recognize the problems and are receptive to implementing the solutions? What is their adoption frequency?

RQ5. What characteristics influence the emergence of specific problems when developing software for the cloud?

RQ4 identifies which patterns are most adopted in the industry. What drives the adoption of these problems? Do internal company characteristics influence their observation? Can we establish a correlation between company characteristics and the appearance of these problems? Identifying correlations between company characteristics and recurrent cloud-related problems will enable companies to define a cloud adoption strategy that can preemptively prepare them for those problems before negatively impacting their operation. To the best of our knowledge, no research correlated company characteristics with the observation of cloud problems and their solutions.

4.3 Research Strategy and Methodology

Zelkowitz and Wallace [MD98a] categorized the experimental models for validating technology into four categories. Quoting from their original paper:

Scientific method. Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.

Engineering method. Engineers develop and test a solution to a hypothesis. Based on the results of the test, they improve the solution until it requires no further improvement.

Empirical method. A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.

Analytical method. A formal theory is developed, and results derived from that theory can be compared with empirical observations.

We structured this research into three stages: preliminary research, pattern mining, and validation. We use the engineering and empirical methods for achieving complementary results. To research our hypothesis while addressing the previously identified research questions, we use the following strategies:

RQ1. What are the recurrent problems when developing software for the cloud?

We start by identifying the challenges from cloud applications in Sections 3.1.1 and 3.1.2 (pp. 26 and 28), with a review of the current state of the art for cloud software design practices and discussing their intricacies. We argue that the current body of knowledge fails to provide reproducible guidelines for designing cloud software. We move on to demonstrate how engineers are approaching these problems in Chapter 5 (p. 55). We begin by designing a reference implementation for a cloud application that orchestrates secure cooperation between sensors and services with limited data and restricted permissions. We then identify a set of practices, systematically captured as a pattern catalog built from interviewing Portuguese startups whom we have inquired about their cloud development and operations practices. We conclude demonstrating that these patterns are useful in the industry by employing participant observation for two weeks, during which we

were able to measure improvements in key metrics in the evaluated company due to the implementation of some of those patterns. Chapter 5 (p. 55) provided the opportunity to experiment with cloud technologies and get acquainted with the industry's cloud status quo, gathering the expertise to continue this research. Using literature research and experimentation, we then mine these problems into a pattern format, as introduced in Chapter 6 (p. 69) and elaborated in Chapters 7 to 9 (pp. 77, 117 and 135).

RQ2. What strategies are adopted for addressing cloud problems?

To understand how the industry is approaching these problems, in Chapter 5 (p. 55), we describe the interview of 25 Portuguese startups, which we inquire about cloud development and operation practices and write a pattern catalog out of them. Chapter 6 (p. 69) further explores this subject, focusing on critical cloud design decisions we have captured in the form of a pattern language. The pattern language thoroughly describes ten frequent problems of cloud design under three categories: the orchestration of infrastructure and services, cloud software monitoring, and discovery and communication for facilitating collaboration between multiple services that compose a cloud application. The 10 novel patterns from the pattern language are further detailed in Chapter 7 (p. 77), Chapter 8 (p. 117), and Chapter 9 (p. 135).

RQ3. What driving forces influence how strategies are implemented?

Chapter 6 (p. 69) describes which forces influence the solutions captured as patterns. Each pattern presents a frequent problem of software design for the cloud and its conflicting forces. The proposed solution properly balances those forces, providing a reliable implementation of the solution for the problem identified in the pattern. The detailed pattern description from Chapter 7 (p. 77), Chapter 8 (p. 117), and Chapter 9 (p. 135) thoroughly identifies the forces that guide the solution for each pattern. We validate these forces by inquiring industry specialists in the case study described in Chapter 10 (p. 149), which enables us to gain confidence in the forces we have identified, just as well as identifying new ones that we can use to iterate and improve our patterns.

RQ4. Are companies that develop software for the cloud aware of these problems and adopt the identified solution?

While interviewing five local startups, we learn if they recognize these problems and which strategies they solve by applying the pattern language. Chapter 10 (p. 149) describes our findings. Chapter 11 (p. 191) describes how we have inquired over 100

companies about their company and product characteristics and tried to correlate those with the adoption level of each pattern identified in the pattern language. We then discuss the identified correlations and how these can be used as guidelines for when to consider implementing each pattern.

RQ5. What characteristics influence the emergence of specific problems when developing software for the cloud?
In Chapter 10 (p. 149), we interview five companies that are developing their software for the cloud. During each interview, we inquire the respondent about the problems they have been through while developing their cloud application, which we then discuss while considering the company's size and application intricacies. We then expand this research to over one hundred respondents in the questionnaire presented in Chapter 11 (p. 191), acquiring the required data to identify correlations between the characteristics of the respondents' companies and the cloud problems they have faced.

Table 4.1 (p. 52) summarizes how we address these research questions along in this dissertation.

ID	Research questions	Related chapters
RQ1	What are the recurrent problems when developing software for the cloud?	Chapters 3 and 5 to 9
RQ2	What strategies are adopted for addressing cloud problems?	Chapters 5 to 9
RQ3	What driving forces influence how strategies are implemented?	Chapters 7 to 10
RQ4	Are companies that develop software for the cloud aware of these problems and adopt the identified solution?	Chapters 10 and 11
RQ5	What characteristics influence the emergence of specific problems when developing software for the cloud?	Chapter 11

Table 4.1: Fundamental research questions in this research and where they are addressed in this document.

4.4 Summary

This chapter identifies the questions that will drive this research. We argue that there are categories of problems while developing software for the cloud that recurrently challenge engineers who lack appropriate resources to address them. With this work, we propose

to explore those categories, their problems, and their solutions. We propose to capture this knowledge as a pattern language. We subsequently propose to validate them through industry interviews and questionnaires that can assert the awareness and relevance of these patterns in the industry.

Chapter 5

Preliminary Studies

This research aims at supporting software engineers while designing their cloud software. To do so, we felt the need to become experts at it, going beyond what Chapter 3 (p. 25) identifies as state of the art with practical experience. This chapter describes two exploratory pieces of research. Section 5.1 (p. 55) describes the implementation of a reference cloud architecture for a Portuguese research project regarding Ambient Assisted Living, Ambient Assisted Living for All (AAL4ALL). Section 5.2 (p. 62) describes a survey held with the Portuguese startup community, aiming to understand the practices and tools they adopt in their cloud adoption. A pattern catalog resulted from this survey, which we validate with a startup to measure the achievable improvement from applying the patterns in an industry environment.

5.1 Experimentation with Cloud Architectures

AAL4ALL[1] was a Portuguese e-health research project held during 2011 and 2015 with a consortium of 32 Portuguese partners from industry and academia, aiming at the development of an open AAL ecosystem by providing an infrastructure for third parties to integrate their sensors and services.

[1] Learn more about the AAL4ALL project at http://www.aal4all.org/.

The project aimed at helping seniors (or care receivers) increase their independence, by instrumenting their house with a set of sensors that could help their daily life. Several use cases were implemented. As an example, the door and window sensors could warn the care receiver about them being left open by mistake, preventing the security risk. Another successful use case was to track Alzheimer patients continuously, providing an alarm mechanism they could trigger if they ever got lost or confused, both in and outside their home.

AAL4ALL required a generic software platform to orchestrate the information that flowed in the ecosystem and to manage and identify users and services [Far+13; Far+14]. We contributed with the design and implementation of a reference architecture that supported the cooperation between services and devices developed by any partner from the consortium. Given the medical nature of the data processed, message passing required a centralized communications system to ensure the adequate authentication, routing, and security of all data exchanged.

5.1.1 Project Overview

Figure 5.1 (p. 57) provides an overview of the project architecture. We contributed to this project with the design and implementation of *Ambient Assisted Living Message Queue (AALMQ)*, a cloud application for orchestrating the communication between any other two entities in the project, for example, a heart rate sensor and the patient's dashboard.

AALMQ provided a message queue where third-party services and devices exchange data, subject to authentication and authorization. There were restrictions on what information subscribers could receive, so we enforced the access to queues and what message could be exchanged in each queue, so that only authorized publishers and consumers could interact with a given queue, preventing data leakage. AALMQ was built around RabbitMQ[2], which provided many of the messaging functionalities. We had to implement a companion service that kept RabbitMQ configured with the queues and individual permissions that were configured in the ecosystem.

Figure 5.2 (p. 58) demonstrates the scenario where a Care Receiver (patient) owns a scale, an electrocardiography reader (ECG), and an AAL Home Gateway at home. Somewhere in the cloud, an AALMQ node assures the communication between the AAL Home Gateway and the Monitoring Web App and Monitoring Mobile App used by a Healthcare Provider and a Healthcare Informal Provider, respectively. The Healthcare

[2] RabbitMQ is an open-source messaging system implementation. Learn more at https://www.rabbitmq.com/.

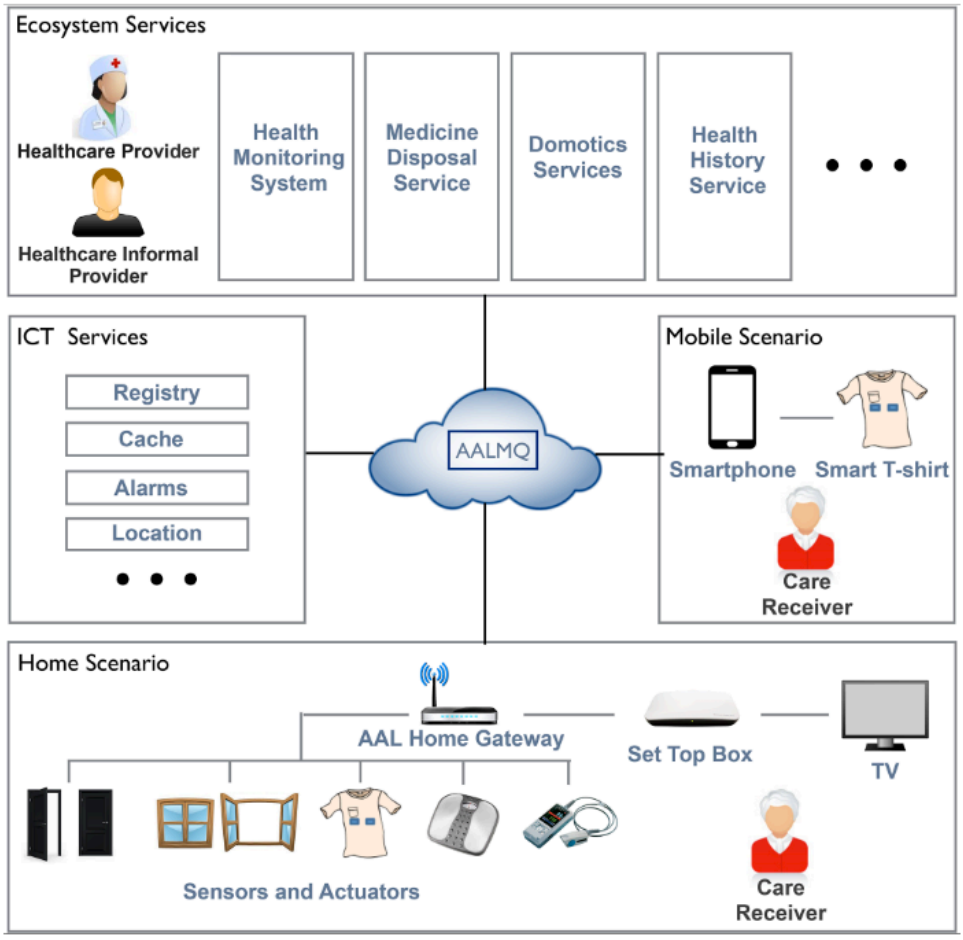


Figure 5.1: AAL4ALL project architecture [Far+14]. The ecosystem services enabled caretakers and health professionals to interact with the care receiver. The ICT services enabled project-wide functionalities; such keeps a care receiver location history. The care receiver had a mobile and home scenario, where he would be monitored by different sensors that would enable notifying the caretaker in case of emergency.

Provider represents an entity (doctor, nurse, group, or organization) that is responsible for monitoring and responding to any problems that may arise with the Care Receiver. At the same time, the Healthcare Informal Provider is a non-specialized person (family, friend) that wants to stay informed about the status of the Care Receiver.

It is possible to identify different communication scenarios, one of which illustrated in Figure 5.3 (p. 59). This scenario describes the mechanism for assuring that a Healthcare Provider (p) receives an alert when the heart rhythm of a Care Receiver (r) is outside a specified range. This description that all participants were previously set up and had permission to exchange messages. The scenario works as follows:

- The Healthcare Provider (p) starts by interacting with the Monitoring Web App, through its user interface, to request the monitoring of the heart rhythm signal of the Care Receiver (r) with a specific period (t) and control range (min, max). This

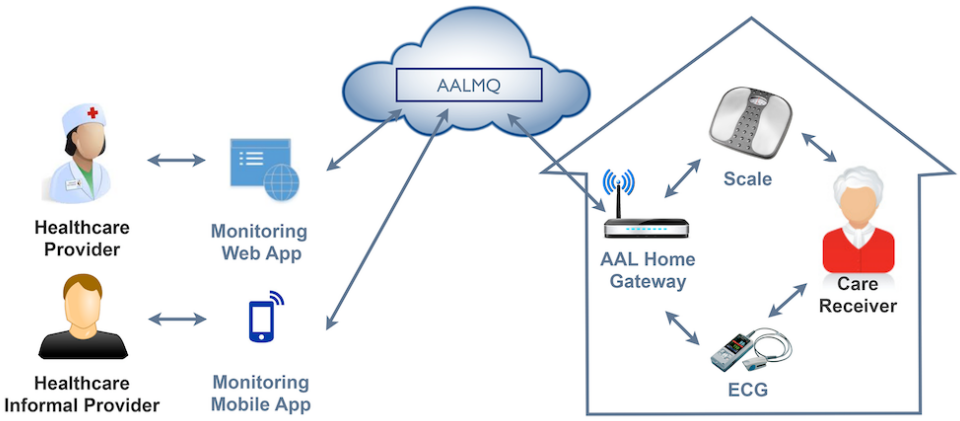


Figure 5.2: Visualization of the intervening components for an Electrocardiography (ECG) reading use case in the home environment of the care receiver [Far+14].

interaction is abstractly represented by the MonitorSignal message in the figure.

- The Monitoring web application prepares a RequestData message to be sent to the AAL Gateway that serves the Care Receiver, having as parameters the type of signal, the identification of the Care Provider requesting the data, the identification of the Care Receiver to be monitored, and the period for data collection. The Monitoring Web App transmits the message via the AALMQ node, by sending a Publish message to the AALMQ node, having as parameters a routing key and the RequestData message. In this case, the routing key is a topic that identifies the target of the message, that is, the Care Receiver. If wanted, the RequestData message may be encrypted by the Monitoring Web App, becoming completely opaque to the AALMQ node.

- Assuming that the indicated AAL Home Gateway previously subscribed to the topic that identifies the Care Receiver r, the AALMQ node forwards the RequestData message to the AAL Gateway.

- The AAL Home Gateway then decrypts (if needed) and interprets the received message. Based on configuration information, it first determines the device that measures the requested signal (hearth rhythm) from the requested person (r). Then, as indicated by the loop interaction operator in the figure, it periodically requests (with period t) a reading from the device, which reports back the measured value (x). For each value reported by the device, the gateway prepares a ReportData message to be sent to the requester, having as parameters the type of signal, the identification of the Care Receiver being monitored (r), the identification of the Care

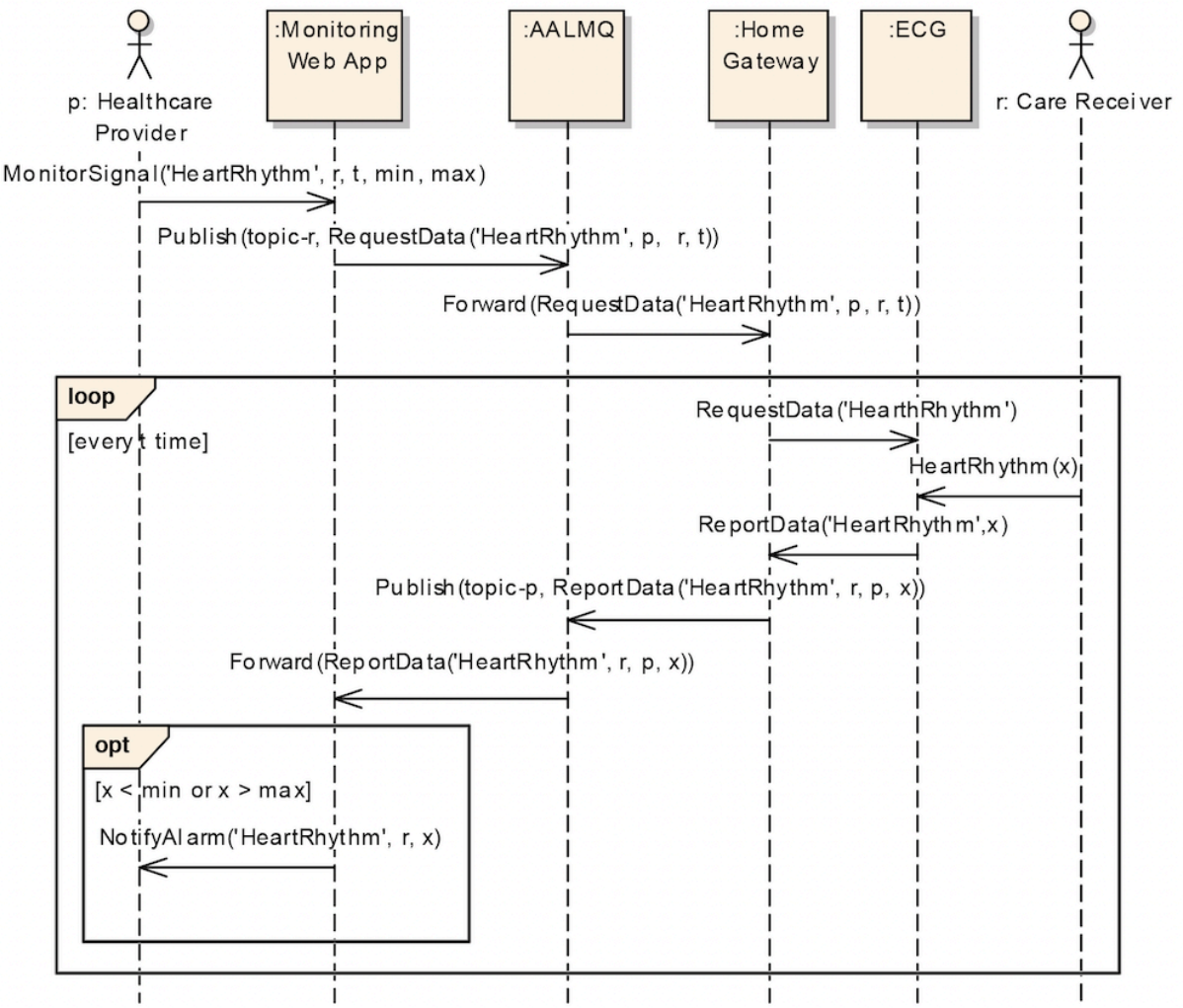


Figure 5.3: Sequence diagram demonstrating message passing between cooperating services in the use case of capturing a patients ECG reading [Far+14].

Provider that requested the data (p), and the actual measurement (x). The gateway transmits the message via the AALMQ node, by sending a Publish message having as parameters a routing key and the ReportData message. Like in the previous case, the routing key is a topic that identifies the target of the message, which in this case, is the Care Provider. Again, if wanted, the ReportData message may go encrypted, becoming completely opaque to the AALMQ node.

- Assuming that the Monitoring Web App previously subscribed to the topic that identifies Care Provider p, the AALMQ node forwards the ReportData message to the Monitoring Web App.

- The Monitoring Web App then decrypts (if needed) and interprets the received message. It appends the received value to its internal database for subsequent

consultation and, in case it lays outside the specified range, sends an alarm notification message to the Care Provider, indicating the type of signal, the identification of the Care Receiver (r) and the observed value (x).

The example presented is based on a small pilot case that was conducted to validate and refine the testing and certification approach presented in this paper. A testing infrastructure, comprising reusable test drivers and stubs, was developed to facilitate the implementation and execution of unit test cases [Far+14].

5.1.2 Development Considerations

The use cases identified before enabled us to elicit the requirements to AALMQ. Given that it would orchestrate the message passing between any other two components in the ecosystem, it must have been considered critical in the project. As such, its design, development, and operations needed to prevent downtime. Changes to it, such as updates or a change in the infrastructure, needed to ensure minimal to no downtime, minimizing the impact in the ecosystem.

There were several challenges to ensure that the application met its requirements. Particularly, we had to consider:

Packaging. Moving software between environments manually, such as copying files between hosts, is not practical. At the time, the Docker container technology was being introduced. We used Docker to package the two services that composed AALMQ independently: the message queue software and the companion software that manage access to it. Using Apache Mesos, a cluster orchestration software, we could instruct it to deploy the containers redundantly on the infrastructure without having to access each server manually, rendering deployments trivial.

Infrastructure orchestration. We quickly found that we would have to use multiple machines to operate the application. A single machine was not enough to handle the traffic at peak usage, and we wanted to provide a redundant system that would continue to operate in case of failure from one node. It would not be trivial to operate each machine, despite any automation adopted. We wanted to abstract the underlying virtual machines and somehow prescribe how software operated in them. We adopted Mesos and Marathon as orchestrating technology to deploy Docker images. The orchestrator deployed the two services on all of the three servers that composed the cluster for performance and redundancy. Additional servers could

be added on-demand, as well as services deployed to them by adjusting a simple configuration.

Deployment automation. Initial deployments were triggered manually. We have noticed that this process was time-consuming and required following a strict script. A minor deviation from that script could render the system offline. Being software engineers, we quickly felt the need to implement automation so that the script could be executed by the computer, preventing human error. A small deployment script was implemented and ensured that we had the correct source code version locally, built and published a docker image from it, and then pulled that image in the production environment's orchestrator. At the time, deployment automation frameworks were not widespread, so we have built that automation strategy ourselves.

Availability and resilience. The research project had several field trials, both with artificial and real patients. With AALMQ being a single point of failure for the whole AAL4ALL ecosystem, it had to be designed with strategies that minimized the probability of it becoming unavailable. As long as that node was available, requests coming into the system were forwarded to one of the nodes randomly. Message ordering was not relevant to the project. If the deployed service stopped working, the orchestrator attempted an automatic recovery. If that did not work, an external monitoring system would notify us by email, enabling a fast manual response.

Traffic and scale. Being an early proof of concept running field trials, the volume of data being exchanged through the AALMQ was minimal. The peak traffic will rarely exceed tens of messages per second. A single machine would be capable of managing this volume of data. Redundancy was mostly motivated to provide availability and resiliency than performance and scalability. Nevertheless, the adopted architecture could quickly scale out to new machines to handle additional traffic.

5.1.3 Conclusion

Implementing AALMQ provided an excellent opportunity to experiment with novel cloud technologies and their orchestration. There were three contributions in AALMQ that furthered our cloud knowledge. Mesos and Marathon acted as orchestration manager and abstracted the infrastructure, facilitating service orchestration and scalability. Docker provided service containerization, enabling execution in isolated and portable

environments. RabbitMQ provided a message queue service with restricted queue interaction.

These design decisions coped with the requirements identified, providing a scalable and secure system for routing messages between distributed cooperating services. Most design decisions were aligned with the pattern catalog that we will later describe in Chapter 6 (p. 69).

5.2 A Pattern Catalog for DevOps and Cloud

Experimenting with cloud technologies was insufficient to understand if our perception of the cloud ecosystem was coherent with the best practices adopted by other engineers. Did we make optimal decisions? Did we use the best technologies? How could we further optimize our setup? To answer these questions, it was no longer enough to experiment with technologies by ourselves; we wanted to understand how the industry was tackling the same challenges we did.

For that regard, we designed a survey for capturing this information with the Portuguese startup community. Startups were ideal research candidates as they typically work with limited teams and budgets, having to work very efficiently to succeed. Furthermore, their goal is to typically scale, so their software designs are concerned with such requirements from the get-go.

During the first months of 2016, we interviewed 25 Portuguese startups. Their responses to the survey led to the creation of a pattern catalog of Cloud and DevOps practices with 13 patterns.

The pattern implementation was evaluated using participant observation with a startup that was in an early cloud adoption phase. We evaluated their development and operations practices and measured how the implementation of our pattern catalog improved them.

Carlos Teixeira collaborated with this research as partial fulfillment of his Master's thesis research [Tei16].

5.2.1 Interviewing Process

This research applied the interview to 25 startups. The interview covered the following categories:

Product. The product section would try to understand what the company did and

secondly if there were any special requirements that would influence the company's choices.

Team management. Team sizes, interactions, project management techniques would be analyzed here.

Software delivery pipeline. In this section, we identified if teams did Continuous Integration, how did they handle the creation of environments for each of the pipeline states and what teams did what in each state.

Infrastructure management. We tried to capture how the companies handled their infrastructure. Did they use the cloud? Which processes did they automate?

Monitoring and error handling. With this section, we aimed to understand if the companies were monitoring their infrastructure, how did they do it and, when errors were detected, how were they responding?

5.2.2 Pattern Catalog

The responses to the interview allowed us to identify tendencies that eventually led to capturing a pattern catalog of 13 development and operations patterns:

Alerting. Define a strategy to alert the team when the system is experiencing failures. The whole team can be alerted, or there can be a sub-team responsible for evaluating the alerts during a given period. Members from this sub-team can periodically rotate.

Auditability. Monitor the application's status and log outputs, raising alerts on failures for quicker team responses when needed.

Cloud. Design the product to use cloud computing enables a faster development by adopting cloud services.

Code review. Peer review source code before accepting it into the master branch to minimize errors and share knowledge.

Communication. Define communication channels for the team so that they can easily stay synchronized.

Continuous integration. Adopt a Continuous Integration (CI) system to automatically test and build the system, alerting the development team of failures in this process.

Deployment. Automate environment setup using deployment scripts and product deployment using REPRODUCIBLE ENVIRONMENTS.

Job scheduling. Schedule job execution in the infrastructure.

Reproducible environments. Package software in a portable way between systems.

Scaling. Adopt vertical scaling, horizontal scaling, or increase the product's availability and capacity.

Team orchestration. Adjust development teams size to be smaller than ten members and promote agile processes and communication.

Version control. Adopt a version control system to facilitate cooperation between developers.

Table 5.1 (p. 64) identifies how many companies we have seen each pattern. These patterns are further detailed in the Master's thesis entitled *Towards DevOps: Practices and Patterns from the Portuguese Startup Scene* [Tei16].

Pattern name	Adoption count
Communication	25
Version Control	24
Cloud	21
Auditability	17
Continuous Integration	16
Reproducible Environments	15
Error Handling	15
Scaling	12
Deployment	11
Code Review	11
Team Orchestration	9
Alerting	9
Job scheduling	3

Table 5.1: Pattern catalog adoption from the 25 interviewed companies, sorted by the most adopted.

5.2.3 Empirical Assessment of the Patterns in the Industry

We want to understand what was the actual impact of applying these patterns in the industry. We expected to observe an improvement on those metrics by measuring key productivity metrics before and after applying the patterns.

This evaluation was performed by applying a participant observation strategy with a Portuguese startup, VentureOak[3]. Participant observation enabled the systematic capture of events, behaviors, and artifacts in the social setting chosen for study [MR06]. This methodology has the researcher joining a team in their daily activities as an additional active member of that team. Other than being able to accurately capture all action from the team, the observant also acts as an active team member, further influencing the team to explore his hypothesis [Kaw05].

VentureOak developed cloud software for third parties and had at the time a little over 20 employees. Their projects had a typical duration of three to six months, built by teams of two to six elements.

VentureOak motivation for collaborating in this research was their own goal to improve their cloud strategy, namely: (1) reduce the time spent deploying software by applying automation, (2) reduce the time required to set up new development and deployment environments by adopting programmatically reproducible environments, (3) minimize state variations between development and production machines, and (4) modernize their technology stack.

The experiment was limited in time to two weeks. Employing participant observation, we acted as an active member of the team, helping them tackle their challenges by applying patterns from our pattern catalog. Three, out of the thirteen patterns, were implemented during this time.

The team needed to reduce the time it took to configure new environments and to ensure consistency in their environments, despite if they would be running development, staging, or production environments. REPRODUCIBLE ENVIRONMENTS pattern addressed this problem, which the team approached by adopting container technology, specifically Docker. The REPRODUCIBLE ENVIRONMENTS facilitated the implementation of the CONTINUOUS INTEGRATION pattern, by leveraging pre-built containers to speed up the testbed setup and reduce the overall test execution time.

Finally, the DEPLOYMENT pattern was adopted to ensure a clean environment was created on every deployment, by having the CI process building a container image fit for production, which the deployment process would then use while deploying a production environment.

The implementation of these three patterns enabled the implementation of a trivial Continuous Deployment (CD) system.

Despite the limited experiment duration, it was still possible to observe improvements

[3] VentureOak was a software house from Porto. They are now part of the German professional network Xing.

regarding the automation of deployments and the creation of development environments with the team. This automation also enabled understanding which software version was deployed at any given time, which was not possible previously.

5.2.4 Results

Table 5.2 (p. 66) identifies the metrics captured with VentureOak at the beginning and end of the experiment. The automation of the deployment strategy slightly slowed down the deployment duration as a side effect of always having to run all tests, which, on the other end, increased the team's confidence on each deploy. This automation increased the developer's awareness of issues with their codebase, reducing the frequency at which build errors where observed.

Metric	Initial value	Final value
Deployment strategy (staging)	Manual	Automatic
Deployment duration (staging)	3 to 10 minutes	15 minutes
Deployment frequency (staging)	When needed	On every push
Environment set up strategy	Manual	Automatic
Environment set up time	5 minutes to hours, depending on developer	Usually less than 5 minutes
Ease to make environment changes	Low	High
Build errors frequency	Low	Medium

Table 5.2: Observed performance metrics at VentureOak before and after the participant observation experiment [Tei16].

Setting up new environments, either for new developers, staging, or production, had the most significant impact, done in up to 5 minutes in any case, by pulling the proper Docker image. A substantial improvement for a task that often required several hours by less experienced team members. The frequency at which build errors where observed resulted from an improvement in the build process that increased the errors detected during the software build and was indeed a positive change.

While it is not possible to argue about the relevance of all patterns, we can conclude that these three implemented patterns very positively influenced the development and operations at VentureOak.

5.2.5 Conclusions

This section identifies recurrent problems and practices from developing cloud software by interviewing professionals. We create a pattern catalog of 25 patterns and evaluate how

often these are implemented in the industry. We then evaluate the impact for a company to adopt these patterns by measuring development metrics before and after the team is provided with the pattern catalog. We observe a performance increase in most metrics measured. The exception is development time, which is slowed down due to the execution of tests in the deployment pipeline. Nevertheless, this increased deployment time was seen as positive, as it vastly increased the confidence over the deployed software.

5.3 Summary

In this section, we have described our initial research and contributions to cloud practices. During this phase, we aimed to grasp a better understanding of the technologies, architectures, and practices frequent in the industry, through both extensive literature research, industry interviews, and actual experimentation.

The first contribution was to the AAL4ALL project, a publicly funded research project to facilitate the continuous observation of caretakers by their caregivers. This project had the contribution of 32 partners from industry and academia, each developing sensors or software that would, in some way, improve the ease of caring for these patients. We have contributed with an architecture and reference implementation to orchestrate the message passing between components in the ecosystem, ensuring scalability, security, and privacy.

Secondly, we have interviewed 25 Portuguese startups to understand their cloud technology adoption and practices. A pattern catalog of 13 patterns resulted from that research. Some of these patterns were expanded and became part of the pattern language presented in Chapter 6 (p. 69).

Finally, we have applied participant observation for two weeks at a startup and demonstrated that the application of some of the patterns from the catalog positively influences the development and operations of the companies that adopt them.

During this chapter, we address the following Research Questions (RQs):

What are the recurrent problems when developing software for the cloud?

We experiment with the intricacies of cloud design and development in Section 5.1 (p. 55), where we highlight the importance of packaging, orchestration, automation, and availability in cloud development.

What strategies are adopted for addressing cloud problems?

Yet in Section 5.1 (p. 55), we describe our strategy to design and implement a reference architecture for a cloud application to orchestrate messages between third-party publishers and subscribers. In Section 5.2 (p. 62) we identify 25 recurrent

practices for the development and operation of cloud software, and evaluate how these can improve the development efficiency of a company.

The work described in this chapter resulted in several publications, namely regarding infrastructures for assisted living cloud systems [Pre+12a; Pre+12b], cloud systems for monitoring and detecting patterns from sensor signals [FSM12], testing and certification methodologies for cooperating services in the cloud [Far+13; Far+14], and cloud designs and patterns [SM13; Sou13].

Chapter 6

Engineering Software for the Cloud

The previous chapters of this dissertation reflected on the intricacies of cloud development, detailing why developers failed at building cloud software or why their efficiency is paramount for businesses to succeed in such competing times. Through preliminary research, we have contributed to a cloud project, going through the challenges of designing for the cloud. We have also surveyed local startups to understand how they were tackling their cloud development. We understood that patterns are a generally accepted strategy to capture knowledge and that, while several authors have described cloud patterns, these are often ill-detailed and, to our knowledge, never empirically validated. This chapter introduces a pattern language for engineering software for the cloud. We propose ten novel patterns and reference two others, relating them in a pattern language. We then present how these can be implemented in an example application. The patterns are further detailed in the following chapters.

6.1 Pattern Structure

The patterns described in this work use a structure inspired by the classical pattern structure proposed in *A Pattern Language for Pattern Writing* [MD98b; WF12], which

included the context, problem, forces, and solution. We developed a superset from it, composed by the following sections:

Abstract. A brief description of the pattern and its applications.

Context. The circumstances that result in the manifestation of the problem. By reading this section, the reader would understand what the driver is for the problem. Experienced users will often relate the context with their previous experiences.

Example. Describes a concrete scenario aligned with the context, where the problem is observable, highlighting the intricacies that make it a complex problem to solve.

Problem. Formalizes the problem, detailing why it is complex to be solved.

Forces. Identifies the forces that influence the design of the solution. Forces commonly oppose each other, leaving for the reader to decide how to properly balance them to customize the pattern's implementation to his specific needs.

Solution. Describes how the pattern addresses the problem and describes its implementation details, which often need to be adapted, considering how the forces are balanced.

Example resolved. Describe how the pattern can be instantiated in order to address the scenario described in the example above.

Resulting context. Elaborates on the benefits and liabilities introduced by this pattern's implementation.

Related patterns. Identifies which patterns can be used with or are incompatible with the implementation of this pattern.

Known uses. Pattern should be extracted from recurring solutions to the same problem observed in the wild. The section identifies implementations that motivated the writing of this pattern.

Further considerations. An optional section in the pattern, where additional details are shared, or a discussion is held, elaborating on the pattern's intricacies.

While capturing this pattern language, some considerations were taken to ensure the individual quality of each pattern. Inspired by the evaluation framework proposed by Seidel [Sei17], the following attributes were considered:

Completeness. Is the pattern description complete? [Ale79] A complete pattern provides a level of detail that enables the reader to identify with the problem and implement it.

Briefness. Does the pattern contain more information than what is strictly needed? A brief pattern goes straight to the point, being easy to read and reproduce.

Validity. Is the stated solution valid and with enough known uses described? A valid pattern documents an accepted good solution and justifies it with concrete examples.

6.2 Methodology

The patterns introduced were mined through observation and literature review, heavily inspired by the first two suggest by Iva towards creating a pattern language in *A Pattern Language for Creating Pattern Languages* [II16]. While doing so, we respect the *rule of three*, cf. Chapter 2 (p. 11), providing at least three independent industry applications of every pattern. Every pattern was peer-reviewed by the software engineering community through publication in one of the Pattern Languages of Programs (PLoP) series of conferences.

6.3 Pattern Language

The pattern language presented in this research is composed of twelve patterns, ten of which novel, organized into four categories: automated infrastructure management, orchestration and supervision, monitoring, and discovery, and communication. Figure 6.1 (p. 72) depicts the patterns in the language and elaborates on their relations. The remaining of this section briefly describes each category and its patterns.

6.3.1 Automated Infrastructure Management

This category comprises two patterns that have already extensively described in the literature: AUTOMATED SCALABILITY and INFRASTRUCTURE AS CODE. We have decided to reference them from the work of other authors but still include them in the pattern language, which is motivated by the relevance they have while designing software for the cloud and how relevant they are for some of the other patterns in the language.

Operations can be decisive for a product's success. Managing operations manually is slow, error-prone, and costly, rendering it hard to trace changes and evolve the

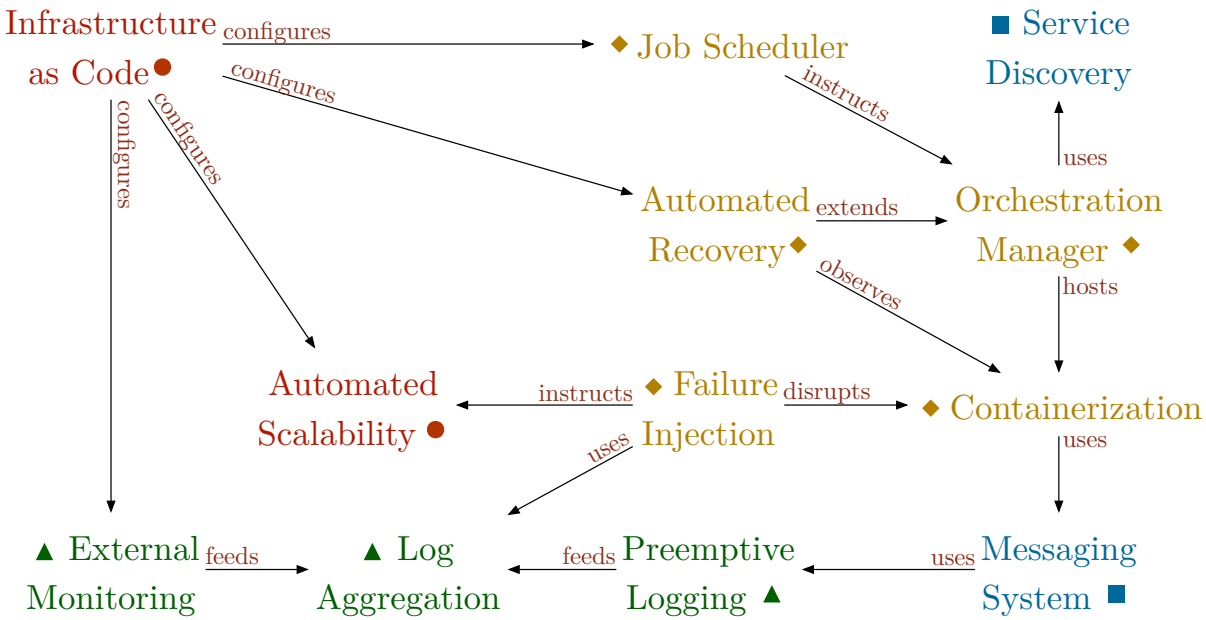


Figure 6.1: The pattern language for engineering software for the cloud, depicting the relations between the patterns (arrows) and the categories that they fall into, *viz*: (■) Discovery and Communication, (◆) Orchestration and Supervision, (▲) Monitoring, and (●) Automated Infrastructure Management.

infrastructure. In order for teams to be efficient, they should automate their operations. We have discussed automated quality testing through the adoption of automated tests and Continuous Integration (CI). Operations should be equally automated, implemented as part of the development process. INFRASTRUCTURE AS CODE enables this practice [Guc17; Dad18; Mor15].

Using a microservice architecture enables the separation of responsibility into multiple smaller services that can be designed, scaled, or orchestrated independently. Multiple microservices can be leveraged to create complex cloud applications [LF14; NS14; Ric17b].

Cloud applications can quickly move from being almost idle to serve millions of requests per second. When developing software for the cloud, keeping up with high traffic peaks is essential to ensure reliable user experience. AUTOMATED SCALABILITY is essential to achieve continuous service performance, by monitoring resource to decide when to scale the system automatically [Wil12; ECN15].

6.3.2 Orchestration and Supervision

After the cloud infrastructure is allocated, software developers need to allocate and operate their software on top of it. This category presents five patterns that help developers to operate their software. These patterns are thoroughly described in Chapter 7 (p. 77).

Traditionally, deploying software in a host coupled it with the operative system, requiring dependencies to be installed and configurations defined that could introduce side effects with other services in the same host. CONTAINERIZATION suggests the usage of containers to package and deploy services in isolation, avoiding that they impact each other or the host.

Deploying and updating software at scale manually is error-prone, slow, and costly. ORCHESTRATION MANAGER can help automate this process by providing a programmatic way to orchestrate services while abstracting the underlying infrastructure. The orchestration can optionally monitor the running services and attempt an AUTOMATED RECOVERY to return the service to a functioning state on failure. Finally, it can also be set to periodically run jobs in the infrastructure, using the JOB SCHEDULER pattern that can be integrated or external to the ORCHESTRATION MANAGER.

With software uptime being critical, developers tend to implement automated recovery strategies, some of which described in the patterns above. With software being software, the recovery strategies themselves are prone to failure and must be frequently exercised to ensure their correctness. FAILURE INJECTION randomly introduces failures in the system to exercise the recovery mechanisms and confirm that they are functional.

6.3.3 Monitoring Patterns

Software running on the cloud can be subject to a vast amount of traffic, which, eventually, and provided enough time, will generate unexpected scenarios. While it is impossible to prevent issues from happening, developers should implement the required strategies to identify when and why issues happen with their software so that they can address them quickly and right from their first occurrence, preventing it from impacting the software again in the future. This category introduces three patterns for facilitating the observation of the application's behavior during execution, which are further detailed in Chapter 8 (p. 117).

PREEMPTIVE LOGGING describes a series of practices that ensure that runtime information is captured and made available for developers to address issues on their first occurrence. It does so by preemptively optimizing the level of logging an application produces. LOG AGGREGATION then describes how these logs should be centralized, for facilitated access, mostly relevant for distributed systems.

The logged information will be most relevant for detecting issues. Automated monitoring strategies are recurrent and essential and can be internal or external to the infrastructure running the cloud software. EXTERNAL MONITOR describes how internal

monitoring is subject to biased observation, incapable of detecting, for example, internet connectivity issues, since it is testing the software from the same local network and proposes the introduction of an external entity to supervise the public interfaces of the application.

6.3.4 Discovery and Communication Patterns

When applications scale, they eventually need to do so horizontally, resulting in the need to deploy additional replicas of the application or, often, to decompose it in multiple services. These services often need to cooperate in providing the application as a whole. Two common strategies to facilitate communication between services are direct communication or an intermediary message passing system. The two patterns from this category facilitate the discovery and communication of services in a cloud environment, using one-to-one or one-to-many strategies, further described in Chapter 9 (p. 135).

The SERVICE DISCOVERY describes how services can discover each other while using an ORCHESTRATION MANAGER, enabling point to point service communication. In some scenarios, point to point communication might not be ideal, namely when multiple instances of a service exist. Those scenarios require a strategy to disseminate messages through multiple instances of a service. This type of one-to-many communication strategies can also address requirements like fine-grained control of what information each service can receive or implement an underlying work queue, with a publisher-subscriber strategy. MESSAGING SYSTEM describes such a strategy.

6.4 Adopting the Language

Resistance to change is, by itself, a pattern. Adopting a pattern language for developing software for the Cloud requires the need for teams to adapt their mindset regarding their organization, processes, and software architectures. While the team must be motivated to change, this pattern language eases its adoption with thorough implementation instructions that elaborate on how to balance the forces observed in a given context. Adoption can also be partial or incremental, adjusting to the team's needs.

This section was inspired by *The Unfolding of a Japanese Tea Garden* by Christopher Alexander [Ale02]. It uses a sequence to describe how the patterns relate and complement each other while describing how they could be implemented in a real-world scenario. We present a sequence of pattern adoptions that describes how the patter language can be leveraged iteratively to achieve a specific goal.

6.4.1 Sequence for a Web Application

Consider the scenario where a cloud *practitioner* needs to create and deploy a redundant Web Application, composed by a client-facing HTTP server and a database. The *practitioner* should design his HTTP server and database as two cooperating microservices. By using CONTAINERIZATION and one service per container, he would create two container images, one of each service. These containers would be highly portable between multiple environments such as local, staging, or production environments, configured using the available environment variables. With INFRASTRUCTURE AS CODE, the *practitioner* would describe the infrastructure required to set up the system. By executing this programmatic description, the required infrastructure would become available. AUTOMATED SCALABILITY can configure the infrastructure to scale horizontally if needed.

To deploy his services in an isolated and scalable way, the infrastructure would be abstracted thought the ORCHESTRATION MANAGER. ORCHESTRATION MANAGER would be responsible for allocating the containers machines in the infrastructure optimally, taking into consideration the total and available resources in each machine. JOB SCHEDULER would be responsible for executing the daily database backup process to an external site.

The web server would use the local network port 12345 to connect to the database, enabled by SERVICE DISCOVERY. The pattern introduces a local reverse proxy on all machines, that exposes a static service port for each service in the cluster. This scenario would not require MESSAGING SYSTEM.

To ensure the service is working correctly, the *practitioner* would have to implement monitoring techniques. EXTERNAL MONITOR service can monitor all Internet-facing endpoints, ensuring that they are both online and responding appropriately. PREEMPTIVE LOGGING can further increase awareness over the system's state by configuring the services with the appropriate logging level. The rationale is that relevant runtime information must be captured for possible debugging porpuses when issues happen, making it very hard to debug them otherwise. LOG AGGREGATION can centralize these logs, index, and make them queryable for efficient usage.

Finally, the *practitioner* needs to ensure his resiliency strategies are enabled and efficient. FAILURE INJECTION can exercise the existing resilience mechanisms by randomly introducing errors in the infrastructure, such as randomly shutting down machines, and verifying that the system recovers automatically.

6.5 Summary

This chapter introduces a pattern language composed of ten novel patterns for designing software for the cloud and two other available in the literature. The language is organized into four pattern categories: automated infrastructure management, orchestration and supervision, monitoring, and discovery, and communication. A hypothetical adoption scenario of the patterns is described using an adoption sequence. The following three chapters further detail these patterns.

With the introduction of this pattern language, we address Research Questions (RQs) 1 and 2, by identifying ten recurring problems of cloud development and their solutions.

Chapter 7

Orchestration and Supervision Patterns

Cloud software requires infrastructure where it is executed. In the past, such an environment required the acquisition of the hardware, setting it all up, including the operative system, installing all dependencies, and then installing the software itself. Today, most Cloud Providers use Virtualization [Sav11; Ace+13; Gra11], enabling the creation and deletion of virtual machines on demand using APIs. Virtual machines are provided as an almost limitless resource, facilitating the allocation of computing power on demand. Platforms for setting up private cloud solutions also exist, enabling the same dynamic allocation of resources on top of private bare-metal clusters using a similar API. This category introduces five patterns for orchestration and supervision: CONTAINERIZATION, ORCHESTRATION MANAGER, AUTOMATED RECOVERY, JOB SCHEDULER, and FAILURE INJECTION.

7.1 Overview

Creating development or production environments manually is a time-consuming process. The probability of error is high, given the commonly large number of dependencies and configurations required. Furthermore, these pollute the host, possibly preventing it from hosting multiple applications. While Virtualization can create a portable environment of the entire hardware and software stack, it always virtualizes the whole hardware and software stack, which is very resource demanding. CONTAINERIZATION is a better alternative, enabling the creation of immutable, reproducible, portable, and secure software execution environments. Containers are considerably more lightweight than full-stack virtualization, as there is no need to virtualize the Operative System layer. Containers prevent polluting the host with dependencies and configurations, making them easier to manage and deploy at scale [BCS15; Sch14]. This approach is also essential for individually scaling each service.

Infrastructure empowering Software in the Cloud is typically volatile and dynamically allocated. As such, orchestration plays a vital role in dynamically identifying the execution setup and adapt the software to cope with it.

Servers in a cluster will differ in hardware details. While some might provide more Central Processing Unit (CPU), others might have higher amounts of Random Access Memory (RAM) available. Not all services are the same. As such, they need to be co-located with the hardware that better meets their requirements. Also, some services need to be co-located in the same host due to multiple reasons, such as latency. Service allocation is not a trivial task. An ORCHESTRATION MANAGER can abstract the underlying infrastructure composed by a varying number of servers with heterogeneous resources and automatically solve the allocation of services to the hardware.

Asynchronous tasks, such as database maintenance, sending emails, or performing backups, are often required to ensure that tasks are being executed at best possible time. These might run at a given frequency or at a single point in time. JOB SCHEDULER can be used to orchestrate the execution of these programs in a cluster and evaluate their result, generating error reports when need.

Software fails [DJG18; PWB07; Gad14; Ser17]. That assumption is even further relevant while orchestrating Software in the Cloud, given its typically large scale. Accepting that it is not possible to prevent software from failing, supervision ensures that services are running as expected, executing the proper action to recover them in case of failure.

Services running inside containers should be resilient in case of failure, providing

AUTOMATED RECOVERY. Exploiting the immutability of containers, the container shall restart itself automatically to try to recover the service whenever it detects a malfunction. Advanced strategies might be applied to recover a service, or set of services, such as restarting a list of services in a specific order. The ORCHESTRATION MANAGER should decide on the best strategy for each scenario.

Mechanisms for improving software resiliency can be built by accepting that software fails. By doing so, developers can hope that the system recovers in unexpected scenarios, but cannot evaluate their confidence in them without testing unexpected scenarios. To ensure reliability and resiliency, a FAILURE INJECTION mechanism can periodically or continuously inject unexpected events in the system, evaluating if it continues to behave appropriately. Fault injection can evaluate reliability by injecting unexpected values into the service and observing if any unexpected behavior occurs. Resiliency can be tested by randomly shutting servers down, ensuring they scale right back up without impacting service quality.

This section introduces the following patterns:

Containerization. Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself. Use a container to package the service and its dependencies and enable its isolated programmatic deployment.

Orchestration Manager. Manually operating software at scale, particularly in architectures that favor microservices and their cooperation, is an error-prone, slow and costly process. Adopt an ORCHESTRATION MANAGER to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements.

Automated Recovery. Services will eventually fail in the long run and need to be recovered in a timely and orderly fashion. Include checks and recovery strategies in the instructions provided to the ORCHESTRATION MANAGER to orchestrate containers, enabling it to monitor and recover failing containers.

Job Scheduler. Short-running jobs need to be scheduled and orchestrated using dynamic infrastructure without permanently allocating resources, possibly requiring ephemeral hardware to execute. Deploy a scheduler service along with the ORCHESTRATION MANAGER that can instruct it to allocate *one time* or *periodic jobs*, releasing their resources for reuse in the cluster when they complete.

Failure Injection. Resilience mechanisms are triggered when the software is failing. Since systems are designed to work correctly, the *status quo* resists to a continuous verification of the correctness of those mechanisms. To ensure resilience, we need to exercise failures to evaluate their impact. Generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms to verify the application's resilience.

7.2 Containerization

Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself. Use a container to package the service and its dependencies and enable its isolated programmatic deployment.

Context

Today's hardware, with multi-core and multi-CPU architectures, is built to execute multiple programs concurrently. Cloud computing often exploits resource sharing for executing multiple services in a single host. Sharing the host's operating system with the hosted services might introduce software incompatibilities between them or quickly clutter the host, as it must mutate its file system to accommodate each service's dependencies. Such introduced the need for isolated environments. Full-stack virtualization quickly became the de facto standard approach to enabling resource sharing, allowing services to be executed in a dedicated installation of the operating system. Paravirtualization further improved that approach by exposing hardware resources directly to the virtualized environment. Still, isolation is achieved with an increased cost of hardware usage required to virtualize the operating system stack on each hosted environment.

Example

Consider a web application that has three services: an HTTP server, a database, and an object caching service. These services share some core libraries, but each depends on different versions. The development team uses a few different Linux distributions for development, but production environments use another. All three services should be deployed on a temporary host for testing purposes and afterward deployed in the

production environment, with the respective configurations. It is not trivial to maintain multiple environments manually while keeping the underlying infrastructure decoupled from the software's dependencies.

Problem

Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself.

Software deployments tend to couple services with their host environment, modifying it according to their needs [Kou18]. When hosting multiple services that share resources, namely file-system, CPU, memory, and network availability, unexpected behavior might be observed as they compete for those resources. Furthermore, situations exist where two services cannot coexist in the same environment due to incompatible dependencies, requiring a dedicated environment for each service.

Forces

The following forces, represented in Figure 7.1 (p. 82), need to be balanced while considering the adoption of this pattern:

Resource Management. Not using all the resources in a server is not cost-efficient, while over-allocating services will degrade their performance.

Overhead. Decoupling services from the operating system might lead to computation overheads.

Supervision. The service status must be monitored, triggering a recovery on failures.

Isolation. Installation of dependencies changes the host, possibly resulting in side effects with other services in the same host.

Portability. Programmatic system deployment requires the packaged software to be easily deployed in different environments.

Configurability. Programmatic system deployment requires a strategy for configuration in execution time.

Security. Different approaches to isolation introduce different levels of security by default.

Solipsism. Each running environment should only manage itself, communicating with external services resiliently.

Persistency. Persist data in the host beyond the service's execution lifetime, possibly being reused in future executions.

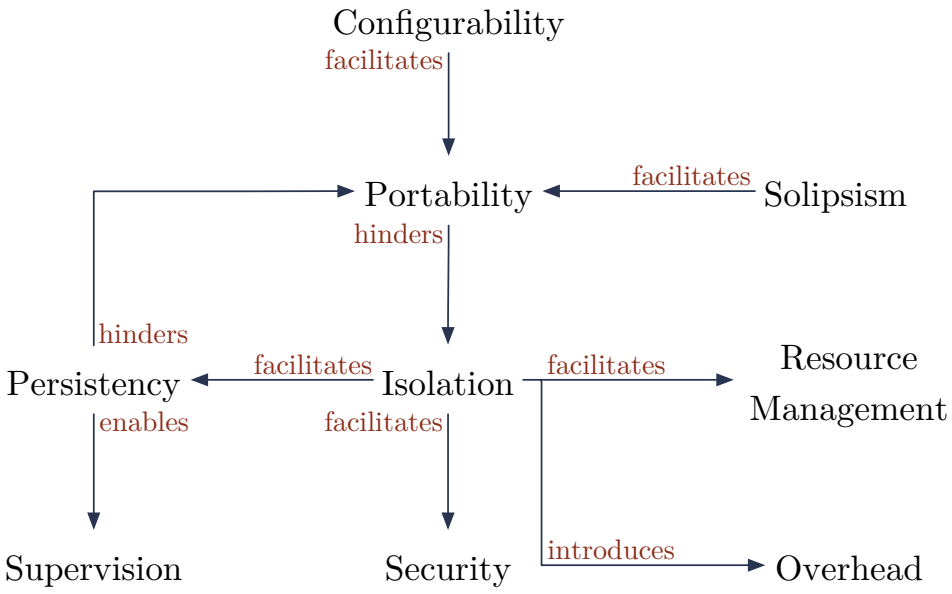


Figure 7.1: Relationship between CONTAINERIZATION forces.

Solution

Use a container to package the service and its dependencies and enable its isolated programmatic deployment.

Full-stack virtualization provides an isolated environment for running software. Despite that, the cost of virtualizing the operating system for each environment introduces considerable overheads in CPU, memory. Portability is also limited, given the increased disk usage. As such, this approach is not an optimal solution for cloud software.

A better solution exists in operating system-level virtualization, also known as containers. A container is a self-contained, isolated environment with a virtual file-system, network, and resource allocation, which is executed within a host operating system [Sol+07].

The container can be created and started programmatically, with configurations provided to the inner software as environment variables, making it portable between hosts. Strict resource allocation ensures that the container will not overuse the available

hardware resources. Figure 7.2 (p. 83) demonstrates how to configure and print environment variables for a container.

Persistent storage can be set up in the container by exposing files or folders from the hosting server inside the container. File system access is limited to those. When the container is removed from the host, all its data is deleted as well. Only folders exposed to the container, if any, are left behind.

On failure, it can restart itself with the same configurations and a clean environment.

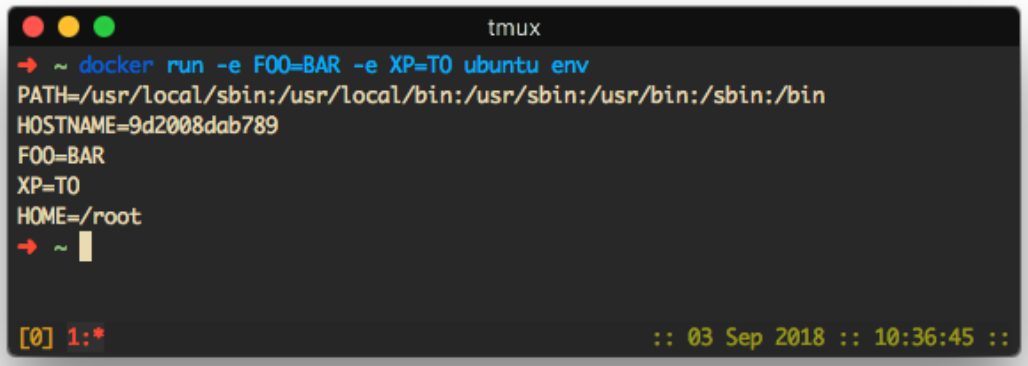


Figure 7.2: Running a containerized Ubuntu image with injected environment variables. Environment variables are provided using the *-e* argument. This example executes the ENV command and exits, which simply prints the environment variables. Environment variables can be read by software running inside the container as a way of providing runtime configurations.

There are multiple container implementations available today, with Docker[1] being the most adopted.

Example Resolved

Each service would be packaged into a separate container. In a development environment, the three containers could be started in the same host. A separate production environment could have each container being executed in an independent host. No changes would have to be made to the containers, other than starting them with the proper configuration as environment variables, which can easily be automated.

If needed, each service can be scaled independently from the others by increasing the number of instances for that specific container.

[1] Learn more about Docker at https://www.docker.com/.

Resulting Context

This pattern introduces the following benefits:

- Resource use is optimized, with overheads being decreased when compared to full-stack virtualization, as only a thin layer needs to be virtualized, improving the performance achievable by a host.
- Resources can be allocated to the container, leveraging the available host's resources between multiple containers, as well as what is exposed from the container to the host and vice-versa.
- Arguments can be provided to the container on execution to configure the service running inside it. Due to its immutability, in case of failure, the container can restart with the original configuration.
- Isolated environment can be easily ported between development and production as the image size only packages the service and its dependencies, leaving out all operating system's components.

The pattern also introduces the following liabilities:

- Paravirtualization is a virtualization technique that exposes part of the host's hardware directly to the virtual machine. In some low-level hardware access scenarios, paravirtualization might provide increased performance.
- Packaging services as containers will still introduce overheads when compared to installing services directly in the host.

Related Patterns

Configuration might be required for a container to be adaptable to multiple hosts and scenarios. Using the Environment-based Configuration pattern, it is possible to use environment variables to configure running services at execution time.

Some containers might have the need to persist information between executions in the host. That is the case of separate databases that cannot lose their data if the machine reboots. With this goal in mind, the Local Volumes pattern may be used to expose a folder from the host inside the container.

Known Uses

Containerization was first introduced in 1982 in the Seventh Edition Unix by Bell Labs, as a tool for testing the installation and build system of the operating system, providing an isolated file-system environment where services could be executed. By 2008 Linux Containers (LXC) was introduced in Linux Kernel version 2.6.24, reducing the virtualization overhead and increasing efficiency [Fel+12]. By 2013 Docker was built, based on LXC, to make containerization easier for a broader audience.

Docker is now the cloud standard for container-based deployment, with native support with multiple cloud providers, such as Amazon Web Services and Google Cloud Platform, both with native support for running Docker containers [Ama15; Goo15]. A draft is being worked on to create a standard format for containers, with RunC being the reference implementation for it, which can also run Docker-created containers [Ini15].

A study by DataDog in April 2018 showed that almost 25% of their clients were using containers and about 50% some sort of ORCHESTRATION MANAGER [Dat18].

Discussion

While container adoption is rising, virtual machines will always be part of cloud computing as the unit of provision of computation. For the development team, the question at hand is if services should be deployed at the virtual machine or container level, their differences, and how to decide. This section sheds some light on this decision. Given the specific context of cloud computing, deploying natively is not within the scope of this discussion.

Providing some context over virtualization, it is built by leveraging a hypervisor to create and execute virtual machines. Hypervisors are responsible for the virtualization of the hardware in a virtual machine and are available in two different flavors: those who run on bare metal, such as Xen, and those who require an underlying operating system such as Kernel-based Virtual Machine (KVM)[2]. In both scenarios, a virtual machine is a fully virtualized computing environment, meaning that every hardware component the virtual machine would see, namely the CPU, RAM, or graphical card, would, be a virtual representation of such element. It is part of the hypervisor responsibility then to map those virtual components to the actual ones available.

Containers work differently, by having the hosted services sharing resources with the host environment, with the actual service execution being managed by the host's kernel, although in an isolated environment.

[2] Xen and KVM are both open-source virtualization servers. Learn more about the projects at https://www.xenproject.org/ and https://www.linux-kvm.org/.

Performance Performance is key in any system. Virtualization efficiency is typically inverse to the overhead introduced by the virtualization system. As previously described, each virtual machine requires its hypervisor to virtualize the hardware and operating system layers, which introduces a large overhead. As such, virtualization is less efficient than containers. Containers provide almost no overhead compared to running in bare metal, given that they share their host's operating system kernel. Theoretically, containers are a much more efficient solution to deploy multiple isolated environments in a server.

This theory has been validated by Xavier, who made an extensive evaluation of native systems performance when compared to three container implementations (LXC, OpenVZ, and VServer[3]) and the aforementioned Xen virtual environments [Xav+13], visually represented in figure Figure 7.3 (p. 87).

Regarding computing performance, Xavier concluded that there were no statistically significant differences between native and the container implementations, but observed a 4.3% overhead with Xen virtualization.

The same study evaluated the memory performance of these three systems and also concluded that containers have similar performance to native, but observed a 31% overhead with Xen based virtualization. We identified this overhead as a product of the hypervisor layer responsible for the virtual machine to native memory address translation.

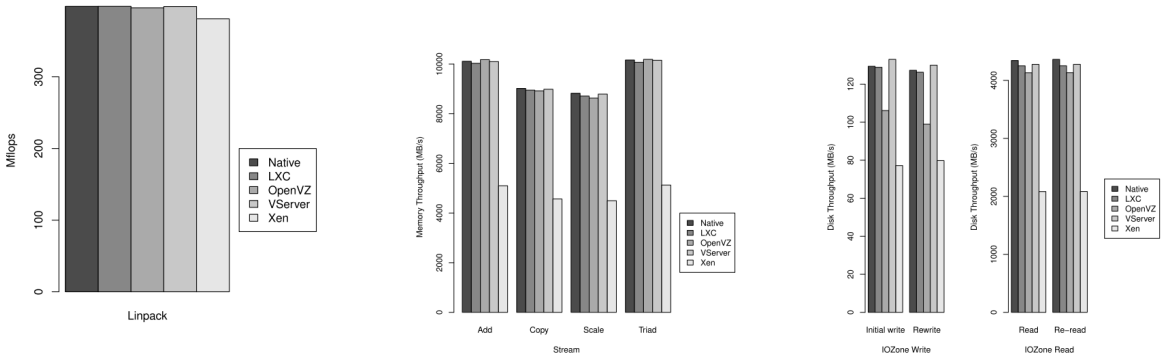
Finally, regarding disk IO, again containers presented a similar performance to native, with OpenVZ outperforming native. Xen, on the other hand, presented poor results with its reading and writing performance about 50% less when compared to native.

Resource Isolation When running multiple virtualized or containerized services in a server, they should not negatively impact the performance of their neighbors. Such is possible by setting hard-limits on resource usage.

With Xen, resource allocation is a requirement for the creation of the virtual machine. These resources are reserved by the hypervisor, which will only expose to the virtual machine the allocated resources.

Containers typically rely on the Linux Kernel Control Groups (cgroups) to enforce resource allocation. Control Groups allow the creation of a resource pool to

[3] LXC, OpenVZ and Vserver are three alternative container implementation. LXC was used internally by Docker until version 0.9, being replaced by lib-container since. You can learn more about these projects respectively at https://linuxcontainers.org/, https://openvz.org/ and http://linux-vserver.org/



(a) Computing performance using Linpack for matrices of order 3000.

(b) Memory throughput using STREAM.

(c) Disk throughput using IOZone.

Figure 7.3: Comparison of (a) computation performance, (b) memory management and (c) disk throughput, from Xavier's work.

be allocated to a given subsystem, enabling resource attribution to those. In practice, it limits the resources available to a service, and its descending processes [Men04].

Enforcing resource limitation introduces an overhead per se, which might have an impact on remaining existing systems. In his research, Xavier ran more than one virtualized or container system, with one trying to use more resources than the ones allocated. He observed that for both Xen and LXC, CPU limitation is effective, not imposing any performance impact on the other hosted system. The same is not valid for memory management, with the Xen hosted service having a minimal 0.9% performance impact, but with LXC presenting an impact of 88.2% [Xav+13]. Several other studies showed similar results, demonstrating that containers introduce negligible performance impact [Sol+07; RD10; Fel+12].

Security Security is essential when executing services inside isolated environments. The service should not be able to access its host unless explicitly configured to do so. Virtual machines, by design, provide optimal security to the host. A service running inside a virtual machine will not be able to understand if it is executing in a native or virtualized environment. Contrary to virtual machines, containers do present an increased security thread. Given that the containerization engine is executed by the host's operating system kernel and that it requires *root* permissions, the kernel itself becomes an attack vector. A set of security measures are recommended for container administrators, namely ensuring that the host's kernel is always using the latest version, that hosted containers are from trusted sources, and that programs within them are always executing using the least privilege possible, meaning that they should only have the required permissions to execute their functions [Mou15].

Flexibility Virtual machines provide the most flexibility for hosts and hosted environments. Given the existence of a hypervisor for a given machine, it will be able to create virtual machines and host any operating system with compatible architecture. As for containers, they currently only run natively in Linux systems, requiring some a virtualization layer in other operating systems to execute. Furthermore, and focusing on the Docker implementation, containers will only Linux as well [Bui15].

Conclusion

We can conclude that containers are still more prone to security flaws than virtual machines. New techniques for securing containers have been made available recently, and more are expected to become available in the future, but it is imperative that the user acknowledges the problem and evaluate its risks while using containers.

7.3 Orchestration Manager

Manually operating software at scale, particularly in architectures that favor microservices and their cooperation, is an error-prone, slow and costly process. Adopt an ORCHESTRATION MANAGER to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements.

Context

Along with cloud computing came complex applications, typically composed of several services that needed to scale out to multiple servers.

Traditional teams would have an operations team that would deploy and operate the software built by the development team. This approach revealed itself impractical due to slow deployments and recurrent conflicts between the two teams [DAC15] due to miscommunication and finger-pointing. To avoid conflict and increase efficiency, DevOps suggested merging both teams, having a single team responsible for the software life cycle. For that to happen, operations needed to be fully programmatic [DAC15].

For achieving this level of automation, abstractions were required to facilitate building

fully automated operation strategies. CONTAINERIZATION played an essential role in enabling the programmatic deployment of software.

Example

An application is composed of two services that need to be orchestrated in an infrastructure with four servers. The service requirements might change with time and must be allocated to suitable hardware. Their current requirements are described in Table 7.1 (p. 89).

Service name	CPUs	RAM	Disk space	Instances	Constraints
HTTP	2	2 GB	5 GB	4	`hostname=unique; location=Europe`
Database	2	8 GB	50 GB	2	`hostname=unique; SSD=true; location=Europe`

Table 7.1: List of services and their possible configurations for a production environment.

The servers might also change with time, with more powerful or specialized hardware being allocated if need. The current servers available are described in Table 7.2 (p. 89).

Server name	CPUs	RAM	Disk space	Server details
Alpha	4	4 GB	500 GB	`location=Europe`
Beta	4	4 GB	500 GB	`location=Europe`
Charlie	4	16 GB	1000 GB	`SSD=true; location=Europe`
Delta	4	16 GB	1000 GB	`SSD=true; location=Europe`

Table 7.2: List of servers available in the infrastructure, along with their resources and meta-data.

Problem

Manually operating software at scale, particularly in architectures that favor microservices and their cooperation, is an error-prone, slow and costly process.

Multiple variants can constraint the allocation of services to servers in an infrastructure. Each service has its requirements, and each server provides a specific set of resources. Furthermore, given the wide adoption of continuous integration and deployment strategies, teams are increasing the frequency at which they deploy their services, with many deploying several times per day [Cyc15], which demands automation in the deployment process.

A common requirement is to ensure that services are allocated to host machines that fulfill its hardware requirements and that this happens without human interaction. Such enables servers to run multiple services while ensuring their execution within the host's resource limits, guaranteeing the expected performance.

Cloud applications can also scale and the infrastructure empowering it must facilitate such scaling as well to adapt to a change in the volume of activity, while optimizing costs.

Forces

The following forces, represented in Figure 7.4 (p. 90), need to be balanced while considering the adoption of this pattern:

Infrastructure decoupling. The development process should not be constrained by the running environment.

Resource allocation. Allocating services without ensuring their requirements will result in unexpected behavior.

Allocation dependencies. Allocating services without ensuring their dependencies will result in unexpected behavior.

Scalability. It must be possible to scale the system either up or down.

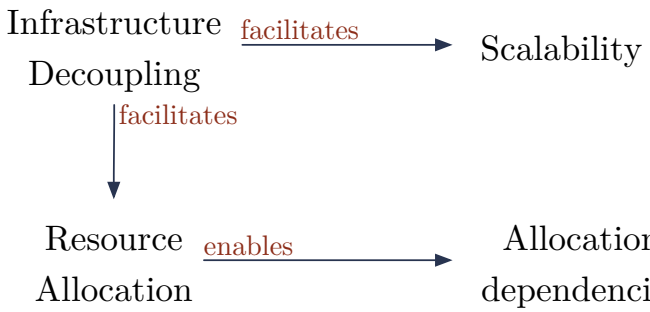


Figure 7.4: Relationship between ORCHESTRATION MANAGER forces.

Solution

Adopt an ORCHESTRATION MANAGER to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements.

Adopting an ORCHESTRATION MANAGER provides abstraction and automation over the orchestration of services. The abstraction is provided by having the ORCHESTRATION

MANAGER evaluating each available server, service, and its requirements and use that information to optimize service allocation. Automation is provided by exposing a programmatic interface that facilitates orchestrating software in the infrastructure.

Services can be deployed programmatically after packaged using CONTAINERIZATION. Using a declarative strategy, the ORCHESTRATION MANAGER can be told what services need to be deployed and their requirements, leaving the responsibility of managing the allocation. Resource allocation is enforced, ensuring that all services are provided their required resources to execute correctly. Most ORCHESTRATION MANAGER enable the specification of additional restrictions such as co-allocations or startup sequences. Listing 7.1 (p. 91) demonstrates how to tell the Kubernetes ORCHESTRATION MANAGER to instantiate two Nginx web servers [Kub].

Listing 7.1 A Kubernetes specification for starting two instances of the Nginx web server.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

ORCHESTRATION MANAGER work using a master-slave architecture, being the master elected automatically and responsible for handling service allocation. Deployment requests can often be issued to any slave, which proxies them to the master [Mes18]. This approach facilitates electing a new master automatically if the current master fails.

Whenever a new slave joins the infrastructure, the master identifies its available resources. When a new service allocation request is received, the master decides where the service should be executed and instructs the slaves to start it. Figure 7.5 (p. 92) illustrates

this interaction. Depending on the ORCHESTRATION MANAGER software adopted, it might be possible to deploy multiple services with a single instruction, with the software being able to automatically resolve service allocation dependencies.

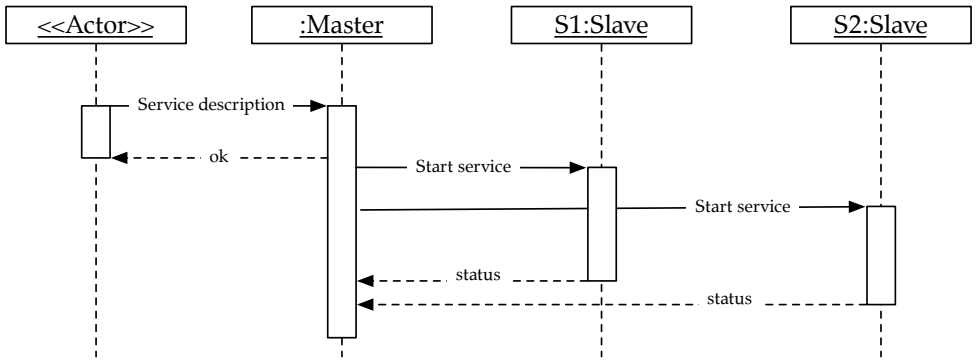


Figure 7.5: Sequence diagram representing communication between master and slaves for service allocation.

If no slave is capable of hosting the service due to a mismatch on the service requirements and those available in the servers of requirements, the master periodically retries the service allocation until it succeeds.

Example Resolved

The team starts by deploying an ORCHESTRATION MANAGER that abstracts four existing servers. By doing so, one of the servers will be automatically elected as master, with the others proxying orchestration requests to it.

A descriptive file can be created for each service, describing how to obtain the respective container and describing its requirements.

Finally, to deploy the services, a request similar to the one from Listing 7.1 (p. 91) is sent to the ORCHESTRATION MANAGER with each service description. The ORCHESTRATION MANAGER master evaluates the resources required by the services and the ones available in each server, instructing the selected servers to deploy the services.

In the example, we can see that the hostname must be unique, meaning that it is not possible to deploy two HTTP or database servers in the same host. Also, the selected servers must be in Europe, with the database service in a server with SSD storage.

Considering those restrictions, the ORCHESTRATION MANAGER would compute a viable solution to allocate the services in the infrastructure, which could be the one identified in Table 7.3 (p. 93).

In this example, when deploying the services, all servers are at full capacity and can fulfill the requested resources regarding CPU, RAM, and disk space. When the service is

Service name	Server name	Applied constraints
HTTP server	Alpha	`hostname=unique; location=Europe`
HTTP server	Beta	`hostname=unique; location=Europe:`
database	Charlie	`hostname=unique; SSD=true; location=Europe`
database	Delta	`hostname=unique; SSD=true; location=Europe`

Table 7.3: The ORCHESTRATION MANAGER can automate service allocation in the available servers. In this solution, two instances of the HTTP server and another two of the database were allocated, respecting the restriction of executing the database using Solid State Drive (SSD) storage.

deployed to Alpha and Beta, the ORCHESTRATION MANAGER subtracts two CPUs, 2 GB RAM, and 5 GB of storage from their available resources, influencing the allocation of services in the future. When deploying the database service, only European servers with SSD storage can be used, resulting in Charlie and Delta being the two only eligible hosts.

Resulting Context

This pattern introduces the following benefits:

Infrastructure decoupling. Service development can be agnostic of the host where the service is going to be placed, describing only its requirements and packaging its dependencies using CONTAINERIZATION.

Resource allocation. Services are allocated in servers that meet their requirements.

Allocation dependencies. Dependencies are respected, managed as constraints for the allocation process.

Scalability. Scalability is achieved by adding slaves to the infrastructure and individually change the number of instances for each service.

The pattern also introduces the following liabilities:

Suboptimal allocation. Allocation using a greedy placement algorithm might result only in a locally-optimal solution.

Single point of failure. In some implementation where the master is not automatically reelected in case of failure, using a single master node would result in a single point of failure.

Related Patterns

Some ORCHESTRATION MANAGER implementations might support additional strategies for running software, but CONTAINERIZATION is the most common strategy.

Known Uses

Kubernetes, by Google, is the fastest-growing implementation of a ORCHESTRATION MANAGER. It abstracts a set of machines, receiving requests for allocating containers in the infrastructure. Kubernetes is under active development, widely adopted, and supported across most cloud providers [Goa16].

Mesos and Marathon together provide another robust solution for achieving the same goal. New services are submitted to the infrastructure using an HTTP Application Programming Interface (API) describing its requirements and constraints. With this information, the master communicates with the slaves, identifying a valid host, and issuing the order for placing the service [HKZ11].

CoreOS offers similar technology, with a centralized registry made available using Etcd [Cor15].

7.4 Automated Recovery

Services will eventually fail in the long run and need to be recovered in a timely and orderly fashion. Include checks and recovery strategies in the instructions provided to the ORCHESTRATION MANAGER to orchestrate containers, enabling it to monitor and recover failing containers.

Context

At the scale that cloud software is operated, it is reasonable to accept that it will eventually fail. Resilience is then an essential requirement while writing scalable cloud software. The development team must introduce the necessary strategies to ensure that the application is functioning correctly or that, at least, it can recover back to a functioning state automatically.

This pattern extends the ORCHESTRATION MANAGER [BCS15] pattern, responsible for executing services packaged using CONTAINERIZATION [BCS15].

Example

Consider a web server exposing an APIis running inside a container in an ORCHESTRATION MANAGER. Suppose that the service had a memory leak, which gradually consumed the memory allocated for the service, and thus making the service unresponsive. The ORCHESTRATION MANAGER sees the container running, but it is still unable to respond to requests while it is still executing.

Problem

Services will eventually fail in the long run and need to be recovered in a timely and orderly fashion.

Cloud software is exposed to a variety of stress conditions, from public Internet exposure to dynamic cloud infrastructure. As such, software should be designed with resilience in mind to ensure it can recover from failures.

With a traditional operations approach, a team member is responsible for identifying failures and deciding the best action to recover a failing system using the defined recovery protocol. This approach is troublesome as it requires manual intervention, which is slow and prone to errors.

Forces

The following forces, represented in Figure 7.6 (p. 96), need to be balanced while considering the adoption of this pattern:

Resilience. Failing containers should recover to a healthy state when failure is observed.

Reliability. Monitoring strategies that are prone to false positives can trigger an unnecessary service recovery.

Automation. Requiring manual intervention for recovering a failing service is error-prone, slow, and costly.

Solution

Include checks and recovery strategies in the instructions provided to the ORCHESTRATION MANAGER to orchestrate containers, enabling it to monitor and recover failing containers.

AUTOMATED RECOVERY is available in most ORCHESTRATION MANAGER implementations [Mes17; Kub18b]. The development team implements health checks for

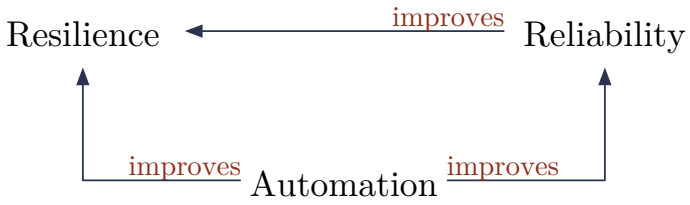


Figure 7.6: Relationship between AUTOMATED RECOVERY forces.

each container to verify if its service is behaving correctly. Most implementations provide at least plain TCP and HTTP checks. The health checks can be provided along with the service description directly to the ORCHESTRATION MANAGER.

To implement recovery strategies, the team needs to evaluate each service individually, deciding which check can be used to identify that the service is failing, how many times each check needs to be retried, and how much time to wait between executing the checks and considering a service as failing. A recovery protocol must be made available along with the health checks to be automatically executed by the ORCHESTRATION MANAGER to attempt the service recovery. Health checks and recovery protocols need to be considered part of the service's development process.

Health checks will be particular to the service running in a container. These might range from checking if a port is receiving connections in the container, to more advanced HTTP-based checks, to executing a command inside the container and monitoring its exit code.

TCP checks verify if a network port is open and accepting TCP connections. These are typically binary checks that validate the service's ability to receive connections.

HTTP checks are more advanced than their TCP counterparts since they can make HTTP requests and validate the HTTP return code and body, making way for more advanced tests.

Developers can implement dedicated health checking endpoints to be queried by AUTOMATED RECOVERY, providing responses that can be easily interpreted to verify the service's status.

While deciding on the supervision strategy, the team needs to evaluate how to prevent false positives. A false positive might be a momentary request that fails, followed by regular service operation. Failing to identify it as a false positive might lead the service to be recovered when it is working.

The recovery operation itself is prone to failure. Implementing this pattern is another step towards improving cloud software reliability, but cannot be relied upon

as unbreakable.

When a failure is identified and triggers a recovery, the ORCHESTRATION MANAGER or adopted AUTOMATED RECOVERY service should log that event and its details. The team can use this log as input for improving their software or to configure notifications to be aware as soon as they happen. This is relevant because restarting the container might only be a temporary solution or incapable of fixing the problem.

Listing 7.2 A Marathon service description, describing the health check policies for AUTOMATED RECOVERY.

```
{
  "id": "toggle",
  "container": {
    "docker": {
      "image": "busybox"
    }
  },
  "cpus": 2,
  "mem": 32.0,
  "healthChecks": [
    {
      "protocol": "HTTP",
      "path": "/health",
      "portIndex": 0,
      "gracePeriodSeconds": 5,
      "intervalSeconds": 10,
      "timeoutSeconds": 10,
      "maxConsecutiveFailures": 3
    }
  ]
}
```

Listing 7.2 (p. 97) demonstrates how a service can be started using the Marathon ORCHESTRATION MANAGER, configuring an HTTP health check that verifies the response code from the */health* endpoint. From this example, the parameters used are *gracePeriodSeconds*, which ignores errors for a given number of seconds after the service starts; *intervalSeconds*, which configures the delay between checking the endpoint; *timeoutSeconds*, which configures the maximum time to wait for a response from the service; and *maxConsecutiveFailures*, which defines the number of times the health check can fail before being restarted.

While implementing this pattern, one needs to decide on how to balance:

Interface coverage. We want to ensure the tests are as complete as possible, covering

all application's interfaces while balancing this investment with the available development effort.

Frequency. We want to run the health checks as often as possible while balancing this frequency with the increase in resource utilization.

Accuracy. We want to prevent false positives by confirming issues redundantly, such as those who might result from a temporary slowdown in the system, automatically recovered inside the container.

Example Resolved

While deploying a service with an ORCHESTRATION MANAGER with support for AUTOMATED RECOVERY, the service definition specifies the set of health checks used to verify the service's status.

During execution, if a health check identifies a problem with a container, the respective container is restarted automatically. While the fundamental issue might persist, the service will once again become available without team intervention. Given the notification sent to the team, they will be immediately aware of the issue and can focus on implementing a proper solution.

Resulting Context

This pattern introduces the following benefits:

Resilience. The ORCHESTRATION MANAGER will be able to recover failing containers automatically.

Reliability. Failing containers will be automatically identified using the implemented health checks, which can have an advanced strategy to prevent false positives.

Autonomy. Failing services are restarted using the implemented recovery protocol so that the system recovers its correct execution state automatically and without requiring manual intervention.

On the other hand, the following liabilities are also introduced:

Relaxation. the development team might disregard software failures since they are automatically recovered.

Unawareness. Without the proper monitoring and logging in place, considering that failing services are automatically recovered, it might happen that the team is not aware of the failure in the system.

Performance degradation. Running health checks against the container will introduce additional load in the system, which might result in performance degradation.

False positives. It might happen that the health checks are not accurate, and the containers restarted while behaving correctly.

Unintended consequences. The service might be improperly designed and unable to be restarted, leaving it inconsistent and requiring manual intervention after a restart. In extreme scenarios, each recovery attempt might further increase the problem. An example of such is when a backup system that is consistently failing during its execution will keep increasing the disk space it occupies without ever having a complete backup until no more space is available.

Related Patterns

The ORCHESTRATION MANAGER pattern describes how containers can be orchestrated in infrastructure automatically, leveraging allocation rules, container scaling, and resource availability. AUTOMATED RECOVERY is commonly related to ORCHESTRATION MANAGER, given that most of its implementations provide some supervision strategy to ensure the containers are working as expected [Mes17; Kub18b].

AUTOMATED RECOVERY enables the automatic recovery of services if their health is degraded. This pattern is essential for implementing FAILURE INJECTION, where the reliability and resilience of the system are tested automatically.

Dynamic Failure Detection and Recovery describes a subset of AUTOMATED RECOVERY, by proposing the existence of a resilient watchdog component that monitors IT resources and, in case of failure, notifies the team and attempts automated recovery [Arc].

Known Uses

Most ORCHESTRATION MANAGER pattern implementations provide AUTOMATED RECOVERY natively, as orchestration and supervision complement each other while deploying services to the infrastructure.

Marathon supports multiple health check strategies. TCP and HTTP are implemented as described in this pattern's solution. Additionally, Marathon supports the *COMMAND*

check, which consists of running a command within a container and evaluate its output [Mes17]. Listing 7.3 (p. 100) demonstrates how a health check can be configured for Kubernetes.

Kubernetes provides a similar approach to AUTOMATED RECOVERY [Kub18b], but with additional features to it. With Kubernetes, developers can set two flavors of health checks: readiness and liveliness. Readiness checks are considered right after the container is instantiated and enable Kubernetes to check if the container is ready to start accepting traffic. Only after the readiness checks pass is the container considered healthy and ready to be used. Liveliness then works as health checks do in Marathon, periodically testing the container for its status, automatically restarting it when unhealthy.

Listing 7.3 A Kubernetes service description, describing the health check policies for HEALTH CHECK using HTTP.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Just like with Marathon, health checks are defined with the service definition, along with the specification of what container to use and how to configure it, as demonstrated in Listing 7.3 (p. 100).

Docker has a built-in supervision mechanism providing a simple restart strategy, which automatically restarts a container either if it fails or when a health check is failing. Health checks can be specified while creating the container [Doc18]. Docker health checks

Listing 7.4 A Dockerfile for building a container based on the Nginx image, leveraging Docker's implementation of AUTOMATED RECOVERY by periodically checking the web server's health.

```
1 from nginx
2
3 HEALTHCHECK --interval=5m --timeout=3s \
4   CMD curl -f http://localhost/ || exit 1
5
6 CMD nginx -g "daemon off;"
```

periodically execute a command within the container, verifying its exit code to evaluate its status. Listing 7.4 (p. 101) demonstrates how to create a Dockerfile that builds a Docker image from the Nginx image and executes Nginx on start, verifying every five minutes if the webserver is responding to requests. If a request takes longer than three seconds to respond, the health check fails, and the container is automatically restarted.

7.5 Job Scheduler

Short-running jobs need to be scheduled and orchestrated using dynamic infrastructure without permanently allocating resources, possibly requiring ephemeral hardware to execute. Deploy a scheduler service along with the ORCHESTRATION MANAGER that can instruct it to allocate *one time* or *periodic jobs*, releasing their resources for reuse in the cluster when they complete.

Context

It is often required that jobs are executed periodically inside an infrastructure managed by an ORCHESTRATION MANAGER. These jobs can range from internal system verifications, maintenance, infrastructure scaling, and many others. These are not long-running services. Hence they do not need to be continuously executing on the infrastructure, as doing so reserves valuable resources that would be idle part of the time.

In a non-cloud context, job scheduling was typically provided by Cron (see Section 7.5 (p. 106)) or similar application. In the context of cloud, Cron is not a viable option, given that it is local to a specific server and not aware of the whole infrastructure. While it can

be argued that it is possible to run Cron in a machine in a cluster, the scheduling would be disabled if that machine failed.

This pattern considers the adoption of CONTAINERIZATION for packaging the jobs to execute and the presence of an ORCHESTRATION MANAGER.

Example

Consider a distributed database, replicated between multiple servers. Despite the replication, keeping frequent backups in a secure remote location is relevant to recover the database from an unexpected scenario in the infrastructure. This backup must happen frequently and automatically, without the team's intervention.

Problem

Short-running jobs need to be scheduled and orchestrated using dynamic infrastructure without permanently allocating resources, possibly requiring ephemeral hardware to execute.

It is common for short-running jobs to be executed in an infrastructure, alongside the hosted microservices. These can vary from database backups to internal system checks. Traditionally, these operations would be the responsibility of the operations team. Some degree of automation could be achieved by leveraging a job scheduler, such as Cron. In the cloud, using Cron is not ideal given that the infrastructure is continuously evolving, that containers are dynamically allocated to their host servers, and that co-location with specific containers or resource allocation rules might exist for running these jobs. Also, using Cron while using CONTAINERIZATION would require a container to be running for the sole purpose of executing scheduled jobs, permanently reserving resources for the container, or using the host's Cron scheduler polluting the host, both less than ideal approaches.

Forces

The following forces, represented in Figure 7.7 (p. 103), need to be balanced while considering the adoption of this pattern:

Automation. Manual intervention is error-prone, slow, and costly.

Frugality. Permanent resource allocation to containers that are idle most of the time is not resource-efficient for the infrastructure.

Reactiveness. Some short-running jobs need to execute as a reaction to an external event (typically called triggers).

Separation of concerns. Short-running jobs are bundled with the description of the resources they require to execute, without needing to know anything about the infrastructure where they will be executed.

Time synchronization. Maintain machine clocks synchronized across the infrastructure to ensure that jobs are started at the correct time, despite what machine is starting the execution.

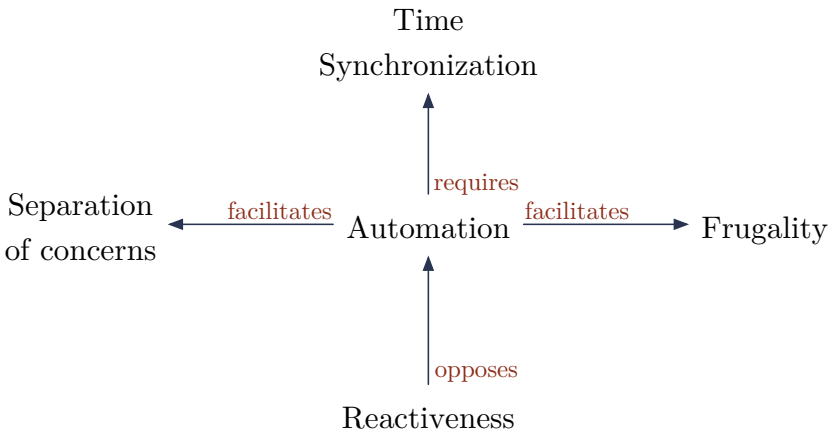


Figure 7.7: Relationship between JOB SCHEDULER forces.

Solution

Deploy a scheduler service along with the ORCHESTRATION MANAGER *that can instruct it to allocate* one time *or* periodic jobs*, releasing their resources for reuse in the cluster when they complete.*

A JOB SCHEDULER extends the ORCHESTRATION MANAGER pattern, responsible for executing services using CONTAINERIZATION, by enabling the scheduling and execution of one time or periodic jobs in the infrastructure. The pattern can be implemented by using a third-party JOB SCHEDULER that already integrates with the adopted ORCHESTRATION MANAGER.

The JOB SCHEDULER service can expose a programmatic, graphical, or both, configuration interface to manage job scheduling. A job specification is composed of the instructions required to execute the job and its resource requirements and schedule details.

The exact information required for executing jobs will be specific to the adopted JOB SCHEDULER implementation, but will typically require the details of a container image to execute, along with a set of environment variables to configure it, supervision criteria

and required execution resources such as the required number of CPU cores or amount of RAM, just like any service would.

The JOB SCHEDULER should integrate with an ORCHESTRATION MANAGER, which is responsible for executing the scheduled job inside a container, honoring its requirement constraints. It works by instructing the ORCHESTRATION MANAGER to execute a container for running the Job, while also providing the requirements for running it.

Allocated resources are freed upon job completion, becoming available for executing other jobs. The integration with the ORCHESTRATION MANAGER ensures that jobs are only started if their required resources are available and restarted in case of an unexpected failure, observed through the job's exit code.

An ORCHESTRATION MANAGER might also provide the possibility for restricting where jobs are executed in the infrastructure, by tagging the available servers and limiting allocation to servers which are tagged with a particular set of labels.

To ensure consistent behavior despite in which node the JOB SCHEDULER is deployed, the hosts should have their clocks synchronized using an external time server.

Example Resolved

Deploy the scheduler service within the infrastructure. The backup operation would be configured in the scheduler to execute every day. The ORCHESTRATION MANAGER would be responsible for ensuring that the container responsible for executing the job is placed in a server that provides the required resources to run the job. It must also ensure the container is co-located with the server running the database, reducing network latency.

A retry mechanism can also be specified, ensuring that the backup job would automatically retry up to a certain number of times in case of failure. If the failure persists, the job's execution is aborted, and the team is notified of the issue.

To ensure that all machines share the same date and time, and jobs are started at the right time, a time synchronization service should be used.

Resulting Context

This pattern introduces the following benefits:

Automation. Jobs are automatically spawned on the infrastructure on their scheduled times, without requiring manual intervention.

Frugality. Resources allocation is minimized for short-running jobs, being recovered by the infrastructure once the job finishes.

Separation of concerns. The scheduled job does not need to know details about the infrastructure, only describe its requirements. The ORCHESTRATION MANAGER will assume the responsibility of placing the container in the right host.

On the other hand, the following liabilities are introduced:

Single point of failure. When the scheduler fails, the ORCHESTRATION MANAGER will not be instructed about the jobs it needs to execute.

Synchronism. Wrong clock synchronization or misconfigured timezones might result in jobs being executed outside their expected times, which might introduce unexpected results.

Reactiveness. This solution does not address reactive job execution.

Related Patterns

Being an extension to ORCHESTRATION MANAGER, choosing a JOB SCHEDULER implementation typically is aligned with the ORCHESTRATION MANAGER choice.

Google also describes how to schedule jobs using their cloud reliably [Goo18]. Using the Chronos JOB SCHEDULER on top of an Apache Mesos ORCHESTRATION MANAGER is explicitly described, as also seen in Section 7.5 (p. 105).

Microsoft describes the behavior for a scheduler pattern [Mic17b], but it only explains how to implement one. This pattern follows a different approach, detailing how to use a JOB SCHEDULER with an ORCHESTRATION MANAGER instead then implementing one from scratch.

Known Uses

Most infrastructure management environments have a companion scheduler service, either bundled in or as a plug-in service.

Chronos is a distributed and fault-tolerant scheduler for the Apache Mesos framework [Chr17]. It exposes an API and user interface with which jobs can be scheduled and monitored. Figure 7.8 (p. 106) shows the Chronos user interface, with four jobs configured. Their state and recurrence are quickly perceived in the status and state column, respectively.

Kubernetes enables job scheduling by making available a built-in scheduler service. Similar to Chronos, jobs can be managed using the user interface or API [Kub17].

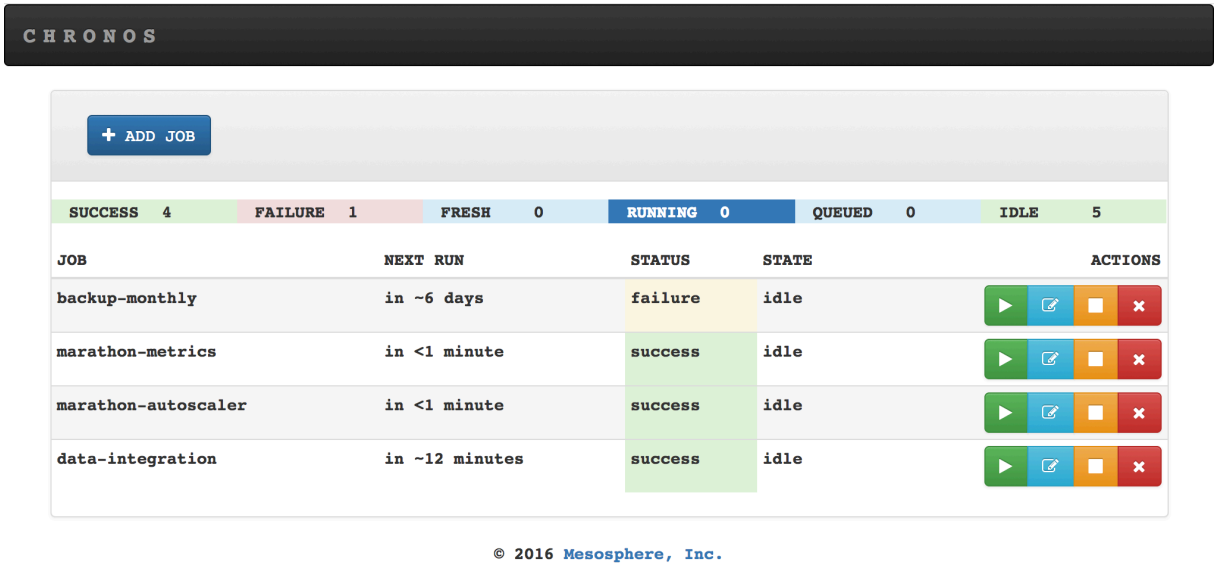


Figure 7.8: The Chronos configuration user interface, showing four scheduled jobs. Chronos enables job scheduling on top of Mesos using a graphical user interface.

Kubernetes API uses the YAML Ain't Markup Language (YAML)[4] format to describe jobs, as demonstrated in Listing 7.5 (p. 107).

Without using a ORCHESTRATION MANAGER, but with a similar objective, cloud providers tend to provide their implementation of a scheduler, which can be used to manipulate their environment or client applications directly [Ama17b; Mic17b]. These typically enable calling the provider's API to start some action, such as running an anonymous function or starting a virtual machine or container.

It was also observed that some companies use a scheduler to periodically evaluate the infrastructure's load and appropriately resize it to cope with the current incoming traffic.

Further Consideration

Most JOB SCHEDULER implementation respect the syntax specified by the Portable Operating System Interface (POSIX) utility Cron, as represented in Figure 7.9 (p. 107) [IO16], for scheduling jobs. This syntax, despite uncommon, has since been widely adopted as the de facto syntax for describing recurrent jobs, as seen in Section 7.5 (p. 105).

While scheduling is a common approach to schedule one-time and recurring jobs, there are other strategies. The event-driven community [Fow17] defends that a reactive is the most efficient way to identify when jobs should be spawned [Bon+14]. With this approach, a JOB SCHEDULER would not be needed, but an additional component to register event

[4] YAML is a human friendly data serialization standard for all programming languages. Learn more at http://www.yaml.org/.

Listing 7.5 Kubernetes configuration for scheduling the execution of a container every minute.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from Kubernetes
          restartPolicy: OnFailure
```

subscription could be adopted, defining which jobs should be spawned after a specific event if observed. For the specific case of time-based execution, this component could react to the clock ticks.

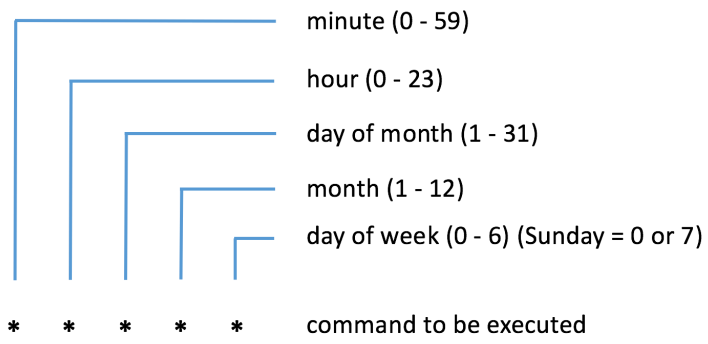


Figure 7.9: Overview of the CRON format, a commonly adopted syntax used to specify the date and time at which a job should be executed and repeated.

7.6 Failure Injection

Resilience mechanisms are triggered when the software is failing. Since systems are designed to work correctly, the *status quo* resists to a continuous verification of the correctness of those mechanisms. To ensure resilience, we need to exercise failures to evaluate their impact. Generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms to verify the application's resilience.

Context

Software fails [Cha05]. This assertion is widely accepted and the motivation for writing resilient software. Application failures can originate both from malfunctioning software or due to external conditions, which might be impossible to predict, such as network failures or defective hardware.

When running software at scale, issues are statistically guaranteed to happen [PWB07]. As such, cloud software must be designed with resilience in mind, meaning that the application should have a set of strategies to recover from problematic situations at both the application and infrastructure layers. Still, resilience strategies are themselves software, hence, prone to failure, limiting the confidence in their efficiency.

Example

Consider an online web application powered by a database. Such a database is essential for the system to work. As such, the database is replicated in hot-standby mode, meaning that the second instance has a complete copy of the first, being used for failover. Furthermore, the database is frequently backed up to an off-site using Amazon Web Services (AWS) Simple Storage Service (S3) and Azure disk snapshots.

Consider now that the second database has an issue and needs to be manually resynchronized. By mistake, an operator manually deletes part of the production database, leaving both inconsistent and lost of production data. When trying to recover the database from the off-site backups, the operator identifies that the backups are not available and identifies that the backup procedure has not been running as expected. No recent backup is available, and the operator will not be able to recover the database to a recent state, resulting in the actual loss of production data.

The example above is a simplified version of an event from early 2017 when a GitLab engineer accidentally deleted part of their production database. Only later, he understood that the existing recovery mechanisms were not properly configured, leaving the system down for over 18 hours and resulting in the actual loss of production data, namely in the changes to projects, comments, user accounts, issues, and snippets. The issue took place between 17:20 and 00:00 UTC on January 31, 2017 [Git17].

Problem

Resilience mechanisms are triggered when the software is failing. Since systems are designed to work correctly, the status quo *resists to a continuous verification of the correctness of those mechanisms. To ensure resilience, we need to exercise failures to evaluate their impact.*

It has been previously asserted that software fails. That was the primary motivation behind the *let it crash* philosophy in the Erlang language and other actor models, where instead of defensively addressing all possible errors, the program was allowed to crash and restarted in an attempt to recover normal execution [Cun14]. The Reactive Manifesto also addresses this type of recovery, with resilience through recovery as being one of the four characteristics of reactive systems [Bon+14].

By relying on software as a recovery mechanism for other software, the recovery mechanisms might fail as well. For that matter, just like any other application, the recovery mechanisms themselves must be validated and frequently tested to ensure their correct behavior.

While designing resilience processes for cloud software, these processes themselves should be monitored, ensuring that the system can recover from a failure properly.

Verifying resilience presents the same problem as verifying software: it is impossible to guarantee that the system is entirely resilient, only that it endures the identified test scenarios. Furthermore, testing software for bugs is more straightforward than testing resilience, as resilience might be influenced by the underlying infrastructure that hosts the application, which might not be under the team's control. As such, resilience testing is not a one-time activity but needs to be continuously improved during the application's lifetime.

At its core, verifying resilience requires the implemented processes to be stressed, putting the application through unexpected scenarios and verifying how well it behaves. This might be problematic by itself if, at some point, the application is unable to recover without manual intervention, rendering it in a degraded state.

This problem becomes more complex as it is insufficient to verify resilience in a staging environment, since resilience is highly influenced by multiple variables, such as the number of allocated resources, how long they have been allocated, or how much load they are handling. While it is possible to create a similar staging environment, even the specific hardware allocated to production might present a different behavior from the staging environment. The only way to increase trust over resilience for an environment is actually to test it.

Forces

The following forces, represented in Figure 7.10 (p. 110), need to be balanced while considering the adoption of this pattern:

Preemptive failure detection. Identify failures in the application before they accidentally impact the application or are exploited by third parties.

Failure generation. Known failures are less likely to impact the system than artificially generated ones.

Resilience. Failure injection might degrade the status of the system.

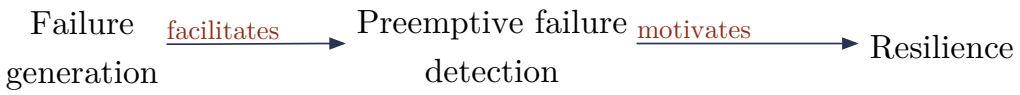


Figure 7.10: Relationships between FAILURE INJECTION force.

Solution

Generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms to verify the application's resilience.

To ensure that the system will recover when a problem arises, its resilience strategies must be frequently exercised, even in production, ensuring that the system does recover to the expected status when a failure happens.

An external piece of software can frequently generate unexpected events at both the application and infrastructure level and monitor how the system behaves, verifying it if it recovers as expected. These events can range from shutting down a container instance to a full virtual machine server. In both scenarios, the resilience strategies should be able to recover the application to the expected status, restarting the container in the first scenario or the machine, and its hosted services in the second.

The adopted strategy used for FAILURE INJECTION should be aware of the application and infrastructure's APIs and randomly inject invalid payloads or shutdown system components.

While implementing this pattern, one must consider:

Completeness. Failures can be injected at both the infrastructure and application level. Only by testing both can we maximize the level of confidence in the system's resiliency.

Frequency. We need to decide how often we will exercise the resilience mechanisms, balancing the resources we are willing to allocate, which directly impacts execution cost, and the level of trust we want to have over the system continuously.

Traceability. We need to understand the impact of injecting failures in the system, by aggregating information from the failure injection system with the infrastructure and application logs, facilitating the evaluation of the impact of a failure injection on the system.

Programatic failure injection. We want to enable the developer to programmatically describe his failures or failure generation logic, so that the failure injection can be automated and executed automatically, reducing the need for manual intervention while running failure injection tests.

There are several attack vectors and liabilities introduced while testing resilience. The following scenarios should be considered:

Application misuse. Generate random inputs to the application's interfaces, including its APIs.

Unexpected load. Suddenly increase the system's load by generating an abnormally high amount of traffic.

Network degradation. Degrade or disable the network to a server, either by disabling the server's network card or use an application that consumes its bandwidth.

Resource depletion. Deplete available disk, RAM or CPU from a server, by starting an application in the server that consumes such resources.

Unexpected component shutdown. Shutting down random servers or other system components, up to disabling entire availability regions.

While exercising the recovery mechanisms with FAILURE INJECTION, the system is expected to be impacted, which should be carefully monitored by observing:

Latency. Some tests will degrade or shutdown resources. While doing so, application latency should be monitored. An ideal resilience mechanism will recover from the injected time without increasing latency above the expected limit. Data from the EXTERNAL MONITOR pattern can be leveraged to observe the application's latency from the user's perspective.

Recovery time. The application should recover within an expected duration. Infrastructural and application logs can be used to verify if a recovery is taking more time than expected, which will introduce the need to improve the resilience mechanisms.

Data. A resilient application should be able to recover from a failure without losing or corrupting data.

Security. During the recovery of the application, the system should remain secure, ensuring that no temporary attack vector is introduced.

Supervision and monitoring patterns such as EXTERNAL MONITOR are companions to FAILURE INJECTION. It is expected that some of the generated events will degrade the system, but its resilience should enable automatic recovery, preventing any impact on the application. If such does not happen, monitoring patterns should identify the degraded system state, providing the required information for the development team to recover the system and, afterward, implementing the required steps to improve its resilience.

It is arguable if FAILURE INJECTION should be applied to production environments, given the risk to degrade them. To prevent impacting production systems, FAILURE INJECTION should first be thoroughly tested in a development or staging environment, being introduced into production when the level of confidence around the application's resilience if definitive. Some teams constrain its execution in production environments to work hours, to ensure failures can be manually managed if the recovery strategies do not function as expected.

While it is arguable if FAILURE INJECTION should be executed against production systems, exercising its recovery mechanisms is the only way to ensure that they are working correctly.

Example Resolved

Adopt a FAILURE INJECTION tool and configure it to generate failures against the application's database and its infrastructure, ensuring that the system can recover automatically.

By exercising the database reliability, the team would have been able to identify earlier that the backup process was not working, just as well as it would be able to understand that the hot-standby replication was not optimally configured. That information would enable them to improve it and ensure the secondary server would be able to sustain the expected level of service required by the application.

Resulting Context

This pattern introduces the following benefits:

Automated failure detection. The adopted tool will generate and inject random events in the system, testing it thoroughly and continuously, identifying issues faster than any manual testing could.

Awareness. Using the EXTERNAL MONITOR pattern, the team can be notified of a degradation whenever a FAILURE INJECTION impacts the system.

Preemptive failure detection. By stressing the application with unexpected events, the team is able to identify failures that could happen at any given time preemptively.

On the other hand, the following limitations will be introduced:

Availability. While testing reliability, it might be the case that an issue is identified and the system's performance degraded. The team should be immediately alerted, take the required actions to recover the system's stability, and implement the required automation to recover from the newly identified scenario.

Resource usage. Exercising resilience will only be possible when resilience mechanisms are available. Often resilience requires redundancy to be implemented, which will always increase the resources required to operate the application.

Unintended consequences. While the system might be able to recover, it might do so while introducing unacceptable consequences. For example, a critical system might lose data during a recovery process.

Related Patterns

When implementing this pattern, SELF HEALING should have been implemented, enabling both the application and infrastructure to recover automatically. FAILURE INJECTION can also leverage LOG AGGREGATION for capturing its action.

The responsibilities for a FAILURE INJECTION tool have been described in the Software Failure Injection Pattern System [LMR01].

Known Uses

Netflix was one of the primary motivators behind FAILURE INJECTION with the implementation of their open-source tool, ChaosMonkey. ChaosMonkey interacts with an AWS account and randomly shutting down infrastructure components. At Netflix, ChaosMonkey is executed against the production environment during business hours, randomly terminating virtual machines. Their rationale is that exposing engineers to failures motivates them to make their services more resilient [Net17]. ChaosMonkey is one of the many tools available in the Simian Army, a set of open-source tools developed by Netflix to help engineers improve their software's reliability [Net11].

Motivated by the impact from the floods of Hurricane Sandy in 2012 in New Jersey, Project Storm is Facebook's approach to resilience testing. At its infancy, it was composed of a set of small drills lead by a reliability team that was designed to replicate the consequences of catastrophic natural events, just like Hurricane Sandy was, by degrading or disconnecting small parts of their infrastructure. By 2014, the team behind Project Storm upped their game, starting to disable entire data centers. The first drills enabled the team to identify several unexpected points of failures [Hof16].

The Principles of Chaos Engineering motivate FAILURE INJECTION. Quoting, "Chaos Engineering is the discipline of experimenting on a distributed system to build confidence in the system's capability to withstand turbulent conditions in production " [Cha17]. In practice, it consists of experimenting with the moving parts of the application, looking for actions that might result in a system failure, such as crashing servers of malfunctioning hard drives.

Further Considerations

Chaos engineering practices are implemented against systems expected to be reliable, validating their reliability. It should be expected that failures are found, and the system should recover without manual intervention. Still, for teams starting to implement FAILURE INJECTION, its execution should be carefully monitored, as some failures might

result in not considered scenarios, leaving the system in an unrecoverable state and requiring manual intervention.

According to the Chaos Community, Chaos Engineer is based on the following principles [Cha17].:

Build a hypothesis around steady-state behavior. Focus on the measurable output of a system, rather than internal attributes of the system. Measurements of that output over a short period constitute a proxy for the system's steady state. The overall system's throughput, error rates, or latency percentiles could all be metrics of interest representing steady-state behavior. By focusing on systemic behavior patterns during experiments, Chaos verifies that it does work, rather than trying to validate how it works.

Vary real-world events. Chaos variables reflect real-world events. Prioritize events either by potential impact or estimated frequency. Consider events that correspond to hardware failures like servers dying, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event. Any event capable of disrupting a steady state is a potential variable in a Chaos experiment.

Run experiments in production. Systems behave differently depending on environment and traffic patterns. Since the behavior of utilization can change at any time, sampling traffic is the only way to capture the request path reliably. To guarantee both authenticity of the way in which the system is exercised and relevance to the currently deployed system, Chaos strongly prefers to experiment directly on production traffic.

Automate experiments to run continuously. Running experiments manually is labor-intensive and ultimately unsustainable. Automate experiments and run them continuously. Chaos Engineering builds automation into the system to drive both orchestration and analysis.

Minimize blast radius. Experimenting in production has the potential to cause unnecessary customer pain. While there must be an allowance for some short-term negative impact, it is the responsibility and obligation of the Chaos Engineer to ensure the fallout from experiments are minimized and contained.

7.7 Summary

This chapter introduced five patterns for orchestrating and supervising services in the cloud. CONTAINERIZATION provided a strategy for isolating and porting services. ORCHESTRATION MANAGER abstracted the underlying infrastructure, automating service placement using containers on top of it. AUTOMATED RECOVERY continuously monitored these services, evaluating if they were or not responding as expected to specific inputs, restarting them on failure. JOB SCHEDULER enabled running transient jobs in the infrastructure at any given point in time and with a configured frequency, releasing the resources once the job completes. Finally, FAILURE INJECTION generates failures in the infrastructure and services, forcing them to recover as a strategy to validate their recovery mechanisms continuously. These patterns facilitate the orchestration of services in the cloud, as well as ensure their continuous execution.

The next chapter introduces the monitoring patterns category, where we describe three patterns for observing service status and state.

Chapter 8

Monitoring Patterns

Monitoring evaluates the state of services continuously, alerting the team when something is wrong. It is an essential practice to ensure the system is behaving as expected. When a failure is detected, the team requires detailed information about what happened, when, and a strategy to facilitate navigation from logs produced by the application. This category introduces three patters for that: EXTERNAL MONITOR, PREEMPTIVE LOGGING, and LOG AGGREGATION.

8.1 Overview

Monitoring can work reactively, by detecting issues using data generated by the application, such as a log file, or actively, by interacting with the services directly and verifying that it is behaving correctly. As such, developers must concern with having their services provide relevant logs while having the service monitored actively. To prevent a biased observation, the application should be monitored from the user's perspective, from outside the infrastructure where it is running, as suggested by EXTERNAL MONITOR.

Teams should adopt PREEMPTIVE LOGGING to ensure their services produce logs with the adequate verbosity, that should be kept for the most prolonged period possible.

Having these logs provides valuable information for debugging the service when a failure is observed.

Having distributed services producing logs will require developers to leverage multiple log files to trace an issue. To cope with the large volume of distributed logs, the team should adopt LOG AGGREGATION, by having a centralized view of all the logs generated by all services in a queryable format.

This section introduces the following patterns:

Preemptive Logging. The information required to debug failures is often lost during their first occurrence due to insufficient log verbosity. Adjust logging verbosity in services and servers within acceptable resource limits, maximizing the probability of capturing relevant information for addressing future issues right from their first occurrence.

Log Aggregation. Services orchestrated at scale produce widely disperse information, resulting in a complicated process to navigate and correlate multiple sources. Aggregate and index all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs.

External Monitor. Monitoring an application from its inner layers results in an incomplete or biased version of the reality. Test the application's public interfaces from an external source, increasing the confidence over the application's status.

8.2 Preemptive Logging

The information required to debug failures is often lost during their first occurrence due to insufficient log verbosity. Adjust logging verbosity in services and servers within acceptable resource limits, maximizing the probability of capturing relevant information for addressing future issues right from their first occurrence.

Context

It is not possible to guarantee that software will behave as expected, so the best bet is to expect the worst. When software fails, information is critical for debugging applications, which makes having execution logs and metrics available from those unexpected scenarios

the most relevant piece of information to understand what, how, and why the software has failed.

Most third-party applications have adjustable verbosity logging capabilities, but first-party applications sometimes neglect that need, causing the developers to lack the required information to mitigate unexpected failures. Given that service cooperation is essential in cloud applications, and considering the uncertainty of the events that lead to unexpected errors, all services should equally generate logs that are the sole resource from developers to understand and mitigate the issue.

Example

Consider a database service in a microservices architecture. The service is responsible for persisting information necessary for other services in the application. At a given point in time, the database crashes. Automated operations practices should ensure that the service is automatically recovered, but, after a while, it crashes again. This behavior is recurrent and without explanation from the development team. The team is expected to identify and fix the issue but is not being able to reproduce it outside the production environment. Without proper information about the production system, the team is rendered incapable of adequately addressing the issue.

Problem

The information required to debug failures is often lost during their first occurrence due to insufficient log verbosity.

Development teams tend to be conservative on their software instrumentation, undervaluing the importance of capturing runtime information. When software fails, it is common that the only debugging approach is to further instrument the software and wait for the issue to repeat itself. This approach decreases the level of confidence in the software quality, as well as requires the team to knowingly leave a bug in their software, given the lack of information to fix it.

Forces

The following forces, represented in Figure 8.1 (p. 120), need to be balanced while considering the adoption of this pattern:

Traceability. Development teams need as much data as possible to be available in order to identify the conditions that may have triggered issues in a service.

Execution resources. Increasing the logging level increases the volume of resources required to execute the service, such as Central Processing Unit (CPU) and memory.

Retention policy. Verbose logging can become expensive to collect and persist for long periods.

Verbosity. Increasing the log verbosity provides additional information for posterior debug, but it also requires additional storage space and human effort to process.

Allocated resource validation. Resources might be over or under-allocated to a service, result in poor usage of the available infrastructure resources.

Privacy. Due to legislation, it might not be possible to persist in some data.

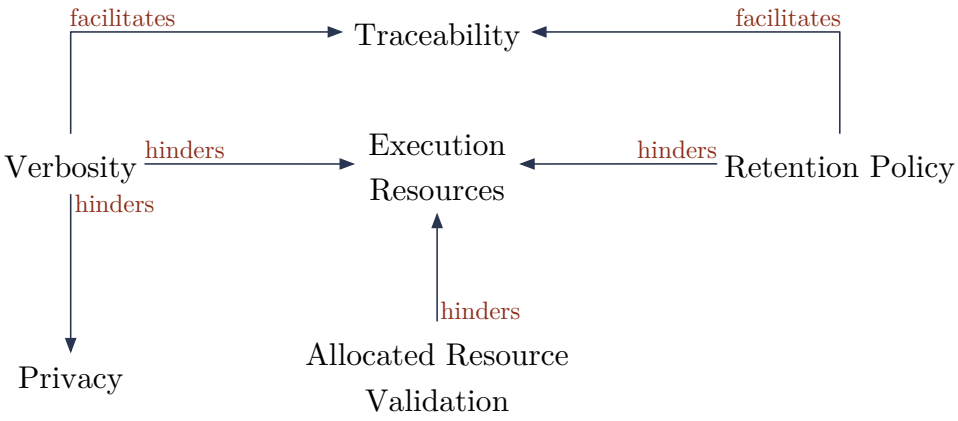


Figure 8.1: Relationship between PREEMPTIVE LOGGING forces.

Solution

Adjust logging verbosity in services and servers within acceptable resource limits, maximizing the probability of capturing relevant information for addressing future issues right from their first occurrence.

Logging is often undervalued by less experienced developers, who are tempted to reduce log verbosity in production to keep the system leaner. By doing so, they unintentionally miss the opportunity of capturing information that would allow them to debug unexpected runtime problems. This may prevent the development team from effectively tackle such problems, unless they begin monitoring the service and server, hoping to observe the issue happening again and capture enough information to identify the reasons behind it.

PREEMPTIVE LOGGING ensures that runtime information from both services and servers is captured, and is an asset available for debugging runtime issues.

Development teams should start by discussing and identifying all the information that can be extracted from the service and respective server. From there, the team should discard the items that will never be useful, setting the optimal log verbosity level for them.

While deciding on what data to keep, resource usage should be discussed, as more information is persisted, the higher the resource impact in the system. A retention policy should also be set as the information becomes less relevant with time.

A scenario where all system events can be captured is ideal, as these can be reproduced in a test environment to debug issues further. Also, once a bug is fixed, they can be replayed in the production environment some types of failures, e.g., one where a specific service is dropping the events sent to it.

Recent privacy trends, such as the European GDPR, might prevent some event data from being physically persisted.

Example Resolved

The team responsible for the database service would discuss what logs and metrics would be relevant to understand how the service is behaving. As an example, they could capture the number of incoming connections, number of incoming queries, query response times, programming exceptions, and the incoming queries themselves. Server metrics would also be captured, namely disk IO, Random Access Memory (RAM), CPU, or network usage.

If the service revealed an issue, they would access the generated log files and use them to understand what triggered it. They could start by understanding if the allocated resources where enough to accommodate the service. If that hypothesis is excluded, they could then dig into the service's logs in order to understand when and why the service started to misbehave.

A retention policy can automatically archive or delete older log entries. When adjusting this policy, the team should allow enough time to ensure the logs are available during the time period when they might be used, preventing them from getting discarded too soon.

Resulting Context

By adopting PREEMPTIVE LOGGING, development teams will gain:

Reproducibility. Service operations can be captured, helping the team understand how it behaved. The whole input stream can be captured and replicated in a controlled environment to understand how and why it reacted in a certain way to a given set of inputs.

Allocated resource validation. Capturing logs from hardware usage from a server will enable the team to understand better how the service consumes resources and optimize resource allocation.

Security and auditing. The development team will be able to trace security problems and threads.

While configuring the service's logging levels, the following should be taken into consideration:

Resources. Should be increased to cope with both the higher CPU and disk space demand of increasing the log verbosity.

Retention policy. Increased retention policy will keep the logs available for a long time period, but such will increase the required disk space required to persist the logs.

Verbosity. Again, the verbosity level should be adjusted to a value that balances an adequate output, with the amount of disk space it will consume.

Security. An attack on this component would expose information from all others.

Related Patterns

The team job is simplified if it is able to query log entries from multiple sources, understand what events were happening in each service. LOG AGGREGATION provides this functionality by moving the logs from their origin to a centralized repository, where they are aggregated and indexed, facilitating their usage. COLLABORATIVE MONITORING AND LOGGING describes the importance of logging and its relevance while deployment software on the cloud [Arc19]. Fernandez described how logs could be leveraged to audit security in the AUDIT LOG pattern [Fer13].

Known Uses

Amazon Web Services' CloudTrail enables the capturing of all Application Programming Interface (API)interactions in an Amazon Web Services (AWS) account, providing complete traceability of all changes through it [Ama17a]. Azure provides a similar service [Azu17]. Spinellis identified log verbosity as a parameter to manually tweak in production when looking for problems [Fu+14]. Fu elaborated on that problematic in his survey [Fu+14], theorizing automated log verbosity adjustment in production as a relevant research topic.

8.3 Log Aggregation

Services orchestrated at scale produce widely disperse information, resulting in a complicated process to navigate and correlate multiple sources. Aggregate and index all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs.

Context

Cloud Computing application can execute at a vast scale, with some teams managing hundreds of services orchestrated on top of thousands of servers. Both the hardware and their hosted services continuously operate and producing relevant information, commonly via log files. Those files must be accessed often, and it is not functional to keep them dispersed in the infrastructure, forcing developers to individual access each machine and respective service to access a given file.

Example

Consider the example from section Section 9.2 (p. 137), where each service is running on its dedicated server. The three services are producing log files, along with the operative system from their host. Imagine now that there was an issue with the AC service or server, rendering the service unresponsive. The developers need to remotely log into the server to access the required log file and debug the issue. Along this process, they understand that the issue was due to a communication error with the messaging service. They now need to access the machine hosting the messaging service in order to debug its log entries.

This process must be repeated for each service and server involved in the issue, going back and forth until the problem is identified. This approach makes it troublesome and inefficient to correlate log entries from different sources and demands that the developer individually accesses each one of the machines.

Problem

Services orchestrated at scale produce widely disperse information, resulting in a complicated process to navigate and correlate multiple sources.

Teams deploying software at scale can easily see their infrastructure grow to tens of servers hosting hundreds of service instances. As suggested by PREEMPTIVE LOGGING,

these services should be verbose at producing logs. At this scale, it is troublesome for developers to leverage these logs, given their sparsity across the infrastructure. The strategy of individually accessing each server and service log file becomes unmanageable.

Forces

The following forces, represented in Figure 8.2 (p. 124), need to be balanced while considering the adoption of this pattern:

Fragmentation. Scattered log files across servers incur in extra effort for the developers to debug the application.

Network propagation. Transferring log data from its source to a central aggregation point requires additional bandwidth and might incur in additional data transfer costs.

Ordering. Propagation of logs through the Internet and unaligned clocks might result in out of order log entries.

Querying. Querying in a log stream is essential to identify relevant information from large collections of logs quickly.

Security. Sending logs across the network should use a secure channel, ensuring that sensitive information is never stolen. The log storage should also guarantee that they are not writable, preventing attackers or other software from changing them.

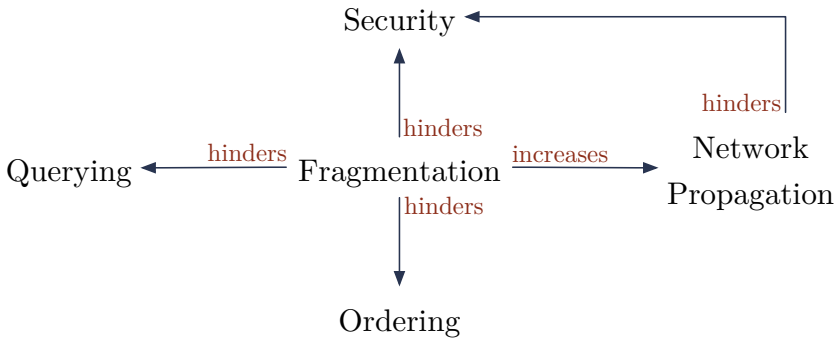


Figure 8.2: Relationship between LOG AGGREGATION forces.

Solution

Aggregate and index all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs.

Having logs available only at their source makes their usage troublesome, requiring the user to log in to each system and either download them or use the set of tools remotely available to query them. At scale, when tracing managed multiple services and servers, this strategy becomes unmanageable and inefficient.

LOG AGGREGATION addresses this problem by providing a centralized system for aggregating and visualizing all logs in an infrastructure. This solution is applied as (1) a log aggregation service is deployed in the infrastructure, enabling the querying and visualization of information from the logs and (2) each service daemon deployed along with it must forward its logs to the log aggregation service. If the team is operating a service not built by them, they can use a log forwarder to read the service logs and send it to the LOG AGGREGATION system.

The centralized log service can persist the log entries in a database, exposing a query interface. Developers can mix and match entries in a single location, facilitating the observation of the infrastructure behavior as a whole, or filtering for a specific service or server. Figure 8.3 (p. 126) represents the relevant components in this process as a class diagram.

A secure channel should be used when sending logs from their origin to this centralized database. Also, it should allow entries from being written but prevent them from being changed, ensuring that logs are immutable.

Example Resolved

Each service and server would send their logs to a centralized log repository service. This service would need to be instantiated in the infrastructure or adopted as an external service. Within it, the developer would have a global view of all logs from all services in the infrastructure. It would be possible to query those logs, filtering them specifically for any specific service at any given time.

Resulting Context

This pattern introduces the following benefits:

Fragmentation. Developers can use an aggregation service to aggregate all the information they need from any service or server in the infrastructure.

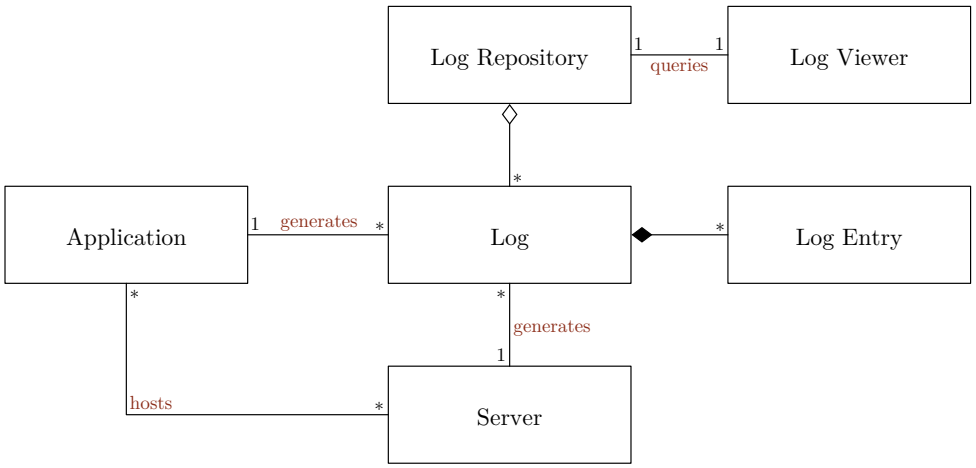


Figure 8.3: Class diagram showing the entities involved in the log generation, persistence, and querying process. Applications and Servers generate log files that are composed of multiple log entries. Each log entry has relevant information about the team to use in the future. Logs are persisted in a remote log repository. This repository aggregates all sources of information, allowing the log viewer to query a single location.

Querying. Once aggregated in a single location, and data can be indexed, allowing developers to query the logs, finding the information they need for their specific task faster.

Security. Communicating logs using a secure channel is essential for keeping sensitive data private. Also, the chosen log storage should be secured to prevent data leakage.

While deciding the technologies to implement this pattern, the following should be taken into consideration:

Network propagation. In order to propagate logs to the aggregating server, additional bandwidth will be consumed.

Ordering. Ordering will rely on the time stamp generated at the server. There might be some errors in cases where the server's clock is not synchronized.

Single point of failure. Without a redundant deployment, a failure in the log aggregation system would revert this system's benefits.

Related Patterns

REPOSITORY describes a generic approach to a data repository [HM]. Fernandez describes the application of log aggregation in the security context to trace user actions [Fer13].

MESSAGING SYSTEM can be used as a communication channel to propagate logs to the log aggregation service. This pattern is further useful if PREEMPTIVE LOGGING is applied in each service in the infrastructure.

LOG AGGREGATION can be used as a source of information for REACT, feeding it with the events used to trigger reactive actions.

Known Uses

Elastic, through their Elastic Stack, leverage Logstash as a tool for acquiring and propagating logs from applications. Logs are propagated to a remote Elasticsearch, a highly indexable JSON document storage. Information can then be queried and visualized using Kibana, a dash-boarding tool for captured data [Ela17].

Loggly[1] is a subscription-based log aggregation cloud service. It provides clients for acquiring logs from multiple platforms and services, making them available in a time-based searchable history.

Roderick et al. have described how their logging service acquired over 50 TB per year, making this data available for over 1000 users daily [RBK13].

8.4 External Monitoring

Monitoring an application from its inner layers results in an incomplete or biased version of the reality. Test the application's public interfaces from an external source, increasing the confidence over the application's status.

Context

While part of the development process is responsible for ensuring resilience, just like it is impossible to ensure complete reliability using software testing [Mal+02], it is not possible to ensure that a system is 100% resilient. Accepting that software will eventually fail is essential to understand the need to increase awareness about the system's status at all times, motivating the need to adopt monitoring on all systems. Frequently, monitoring systems live within the application's infrastructure, which might bias the awareness about the actual state of the application, given all the external variables introduced by using the Internet as a distribution channel.

[1] Details at https://www.loggly.com/.

Example

Consider an authentication service, part of a larger application. Provided with a valid login, it should output an authentication token for interacting with the other services in the application. Consider the scenario where when used from within the infrastructure, the authentication service works as expected, but, when accessed from a remote application, the authentication service is inaccessible. A misconfigured firewall can cause such discrepancy.

This scenario demonstrates that a service can have different statuses depending if it is observed from within the application's infrastructure or a remote site.

Problem

Monitoring an application from its inner layers results in an incomplete or biased version of the reality.

Software failures can be catastrophic to business owners. Application downtime consequences can range from client complaints to loss of confidence in the application and, ultimately, user abandonment or contractual breach. Given the ever-growing offer of online services, a failing application can easily be replaced by a competitor.

In case of failure, the development team should quickly be aware of the application's status, facilitating a quick reaction.

This awareness must not depend on the application or its infrastructure, as that would bias the observation. In the context of cloud computing, simply monitoring the application alongside its execution is biased and prevents the detection of several unpredictable Internet-related issues, such as misconfigured or failing routers, CDN, DNS, or firewalls which would directly impact the client's access to the application.

In the example from Section 8.4 (p. 128), a misconfigured firewall is inadvertently blocking traffic from a valid source, leaving the service inaccessible from the outside. This issue would not be identified by monitoring the application from within the infrastructure, as the firewall would not be used between two internal services.

Forces

The following forces, represented in Figure 8.4 (p. 129), need to be balanced while considering the adoption of this pattern:

Confidence. Maintain awareness of the system's state without relying on its internal information or be biased by internal monitoring.

Recency. Be notified as soon as a possible complication is identified in the application.

Coverage. Confidence level increases with the increase of test coverage, that is, with the increase in the number of public endpoints and their tested features.

Resource usage. Minimize the impact of monitoring on the application's resource requirements, directly impacting performance or cost.

Security. Minimize the attack vectors for the application. Exposing sensitive application details to additional external tools will create a new attack vector.

Geographic description. Running tests from different world locations increases the level of confidence that the system is working worldwide.

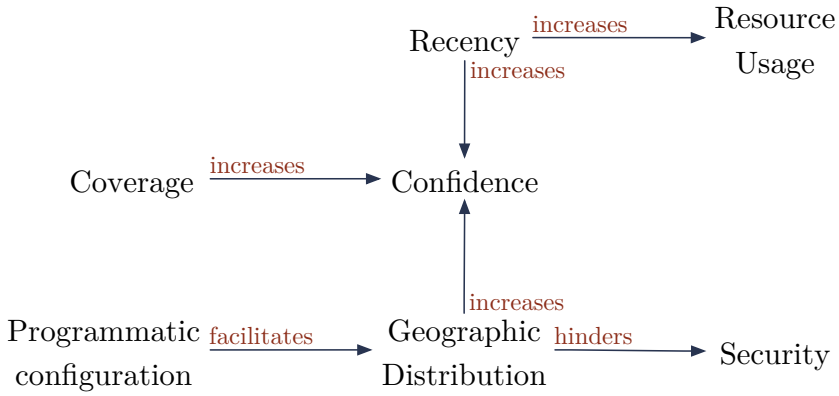


Figure 8.4: EXTERNAL MONITOR forces relationships.

Solution

Test the application's public interfaces from an external source, increasing the confidence over the application's status.

Resilience is an essential requirement of any cloud software. Still, just like with software testing, it is impossible to guarantee that a system is resilient and will not fail. Besides improving the system's resilience, the development team should also invest in awareness of the system's status, reducing the time required to react to a failure.

EXTERNAL MONITOR consists of the frequent execution of tests against the public interfaces of a live production system, evaluating if they are responding as expected. Tests are configured and executed from a service running in a separate network environment from the application itself and run without any knowledge of the application's state (as

a black-box test), providing an accurate observation of the system's status as seen from across the Internet.

Test coverage can range from a basic status check to see if the service is up to having a batch of tests covering all the application's public interfaces and their different uses. Such a level of coverage could be seen as black-box integration tests executed against a live environment. It is up to the team to balance the level of coverage with the intended level of confidence in the system's status.

The team can either develop their EXTERNAL MONITOR tool or adopt one of the many third-party tools available. Developing a tool for external monitoring would require considerable investment in development and operations. On the other hand, adopting a third-party tool introduces a financial cost for using the service, as well as it widens the attack surface to the application, as sensitive information such as user credentials need to be shared with the system. A hybrid approach could consider adopting an open-source tool for doing external monitoring, which will prevent sharing sensitive credentials with a third party while still requiring little investment in developing the software.

Some tests might require sensitive data to execute, such as user credentials. In case of an attack on the monitoring platform, this might hinder security, leaving those credentials exposed. Frequently rotating these credentials can help mitigate this issue.

While implementing this pattern, one must consider:

Recency. We need to decide how often we will run the external monitoring tests, balancing how fast do we want to know when an issue appears with the system as the load introduced by the tests will increase resource usage.

Development effort. We need to balance the completion of test coverage with the time required for developing new tests.

Security. We need to decide which, if any, credentials should be made available in the external system to test protected interfaces, at the cost of possibly exposing sensitive data.

Accuracy. We want to prevent false positives by confirming issues redundantly, such as those who might result from latency or network partitioning.

Geographical distribution. We might want to distribute tests geographically, ensuring the application is working within the specified parameters, despite where the traffic is originated. Geographically distributed monitoring also enables verifying the correct behavior of distribution components such as CDNs.

Traceability. We want to understand why a test has failed by evaluating the inputs and outputs used to identify the failure. LOG AGGREGATION pattern can be leveraged to combine logs from this pattern, as well as logs from FAILURE INJECTION and the remaining components of the system, providing a unified view over the system's behavior.

Programmatic configuration. We want to manage monitoring tests automatically as part of the deployment process, eliminating the need for manual configuration, hence, increasing confidence in the tests.

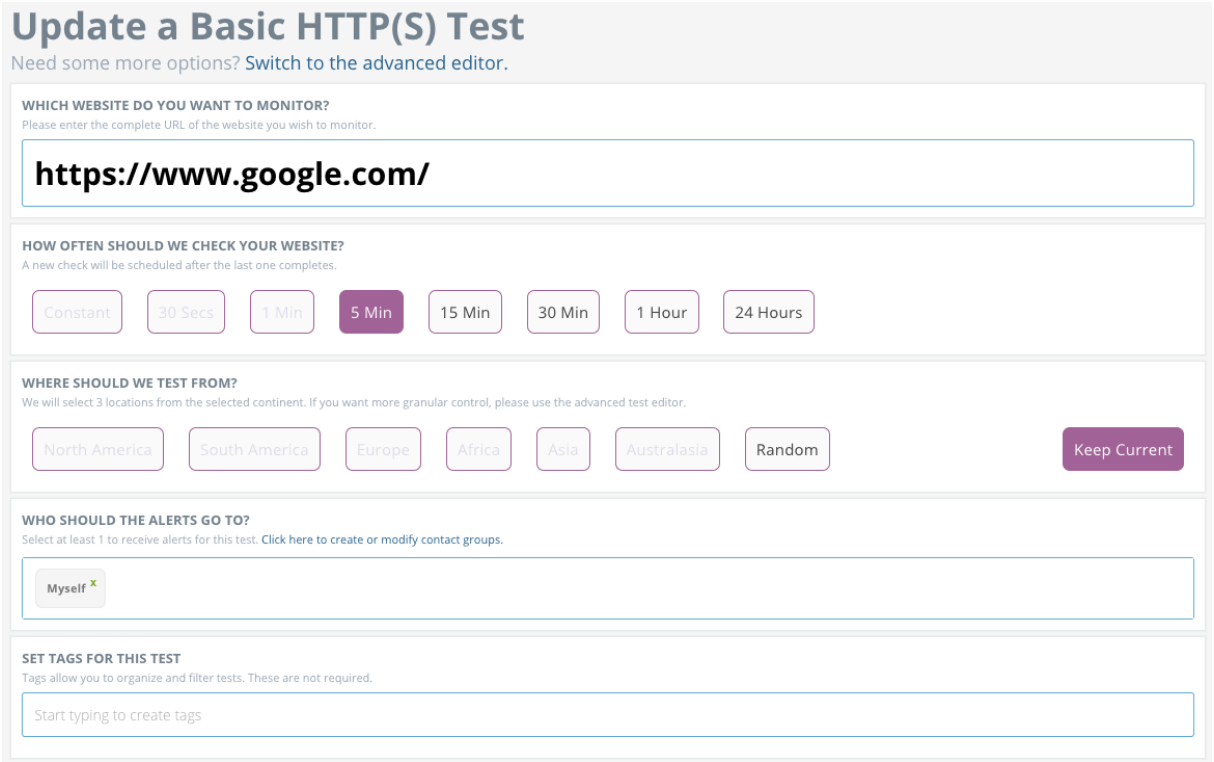


Figure 8.5: Statuscake's HTTP(S) test creation interface, showing a basic HTTP test for Google's homepage, which will execute every 5 minutes from a random server.

Third-party tools for implementing the pattern often allow tests to be created from both a graphic interface, as seen in Figure 8.5 (p. 131) and a programmatic interface. The latter enables tests to be configured as part of the application's deployment process.

EXTERNAL MONITOR is not a new strategy for cloud computing. *Cloud monitoring: A survey* [Ace+13] details thoroughly why monitoring is an essential aspect of cloud applications and describes over twenty tools to implement it, ranging from commercial to open-source offers, making it an excellent guide for selecting the tool used to implement this pattern.

Example Resolved

Considering the example, this pattern would be implemented by adopting an EXTERNAL MONITOR system that would make an authentication request to the authentication system and confirm that the answer contained a proper authentication token. This test would be configured in the monitoring platform at the end of the deployment process, ensuring that the application is tested and working as expected right from the moment the deployment is complete.

Tests would be executed at a configured frequency and from different geographic locations to ensure that the application behaves correctly, despite where a request has originated.

Possibly at a later stage, and for increasing test coverage, any other interface in the service could be tested as well.

To enable interacting with authentication-protected areas of the application in production, a mock user can be set up in the system. This way, in the case of data leakage in the monitoring system, no significant impact would be observed on the monitored application. Arguably, testing against a single user account that was created solely to test the system might provide a bias *per se*.

Resulting Context

By adopting EXTERNAL MONITOR, development teams will gain:

Confidence. Given continuous independent monitoring, there is an added confidence that the system is behaving as expected if no alarm is raised.

Traceability. The team will be able to understand what behavior was observed as a response to any failing request using the EXTERNAL MONITOR logs.

Programmatic configuration. The team will be able to evolve test scenarios along with their development, using the EXTERNAL MONITOR API to setup or update tests.

On the other hand, the following liabilities can be introduced:

Security. When the communication channel is properly secure, no data leakage can occur by executing the tests from a EXTERNAL MONITOR provider. The team must trust the provider. Given an attack against it, sensible information might be exposed. It is the team's responsibility to minimize or eliminate the need for sensitive information such as credentials for executing the tests.

Resource usage. If careless, the team might create a large volume of tests at a high frequency, which might generate enough load to degrade the application. It is up to the team to properly balance the volume of tests and their frequency.

Related Patterns

EXTERNAL MONITOR providers expose APIs, which can be used to manage the tests programmatically. A team that adopts INFRASTRUCTURE AS CODE will be more efficient at managing their tests.

EXTERNAL MONITOR can be used to feed information for LOG AGGREGATION, facilitating a centralized view of the issues observed in the application from this monitoring strategy as well.

HEALTH ENDPOINT MONITORING from Microsoft is similar to this pattern proposes the creation of HTTP health checks exposed by the application, so that an external tool can verify the application status [Mic17a]. That implementation differs from EXTERNAL MONITOR, as it requires specific endpoints to be implemented and tested from the external health checking tool. Instead, EXTERNAL MONITOR proposes that the external tool interacts with the application as a client would, using any public interface, not limited to HTTP, validating that it is providing the expected answers.

The COLLABORATIVE MONITORING AND LOGGING pattern [ECN15] describes how monitoring and logging activities can be coordinated between a cloud consumer and provider, describing that monitoring and auditing requirements can be described by the consumer but observed by the provider. This approach is similar to EXTERNAL MONITOR, given that the monitoring behavior is extracted from the application the consumer is developing and executed with an external tool, managed by the cloud provider.

Known Uses

Multiple services are available, providing the EXTERNAL MONITOR tool required to implement this pattern. StatusCake, Pingdom, or NewRelic [Rel17; Pin17; Sta17] are only three of those applications. Pricing and features set them apart, with most being able to test at the HTTP and TCP layers.

Further Considerations

Juvenal, a first-century poet, in his *Satires* series of books wrote the famous Latin quote "Quis custodiet ipsos custodes?", roughly translated to *who watches the watchmen?* [Win99]. This quote can still today motivate discussion around cloud

monitoring. By relying on an external tool to monitor the system, we are delegating the responsibility of capturing failures to an external system. What must be taken into consideration is that the external system is a piece of software as well, which might also. In such a scenario, a failing system would not be detected since the monitoring system would also be unavailable.

8.5 Summary

This chapter introduced three patterns for observing cloud software status and state. PREEMPTIVE LOGGING recommends that developers preemptively adjust their logging level to ensure they capture relevant information to debug potential future issues with the system. LOG AGGREGATION facilitates working with logs from multiple sources, aggregating them in a centralized platform where developers can slice and dice it for quicker and more meaningful exploration. Finally, EXTERNAL MONITOR recommends the monitoring of the public service endpoints from an external location, ensuring that the system is monitored independently by interacting with the system as a user would, generating alarms for the team on failures. These patterns help developers increase their confidence in the correct operation of the application, providing the required data to dissect issues when they are observed.

The next chapter introduces two discovery patterns that help developers design how their services can cooperate, both synchronously and asynchronously.

Chapter 9

Discovery and Communication Patterns

In a microservice architecture, multiple services need to cooperate in providing the application as a whole. Cooperation requires that the services first discover and create communication channels between them. This introduces MESSAGING SYSTEM and SERVICE DISCOVERY, two alternative strategies for service discovery and communication.

9.1 Overview

While using an ORCHESTRATION MANAGER that dynamically allocates containers, the exact network location where another service is running is unknown. Using a SERVICE DISCOVERY, a service location can be abstracted through a local network port exposed on every machine that is always forwarded to one instance of the service, possibly balancing traffic between multiple instances [Sch+06]. This is easily achieved by preemptively creating a table that maps local ports to services. Whenever the port is mapped, the service is up, and the communication established.

Some scenarios prevent communication from being point-to-point, for example, for scalability reasons. A MESSAGING SYSTEM can be used to deliver messages between microservices, eliminating the complexity associated with service discovery [Gaw02], at the cost of additional latency.

This section introduces the following patterns:

Messaging System. As the volume and complexity of interacting services increase, point-to-point communication channels become unmanageable, hindering fault-tolerance, resiliency, and scalability. Use a MESSAGING SYSTEM, colloquially known as *message queue*, to abstract service placement and orchestrate messages with the optimal routing strategy between them.

Service Discovery. In a dynamically allocated infrastructure, services require a discovery strategy to establish a communication channel. Abstract service network details by relying on an external mechanism that facilitates communication and balances traffic between two services.

9.2 Messaging System

As the volume and complexity of interacting services increase, point-to-point communication channels become unmanageable, hindering fault-tolerance, resiliency, and scalability. Use a MESSAGING SYSTEM, colloquially known as *message queue*, to abstract service placement and orchestrate messages with the optimal routing strategy between them.

Context

The adoption of microservices as an architectural style introduced the need for services to cooperate in a decentralized and possibly unreliable environment. It is not guaranteed that every component is online at all times, nor that each service has a stable IP address (Internet Protocol) or a fixed number of instances running.

These intricacies of cloud computing introduce several requirements. Namely, services need to *communicate* with each other in an ever-changing environment, the communication process must be fault-tolerant, ensuring that the system as a whole is *resilient* when confronted with irregular behavior from either side of the communication, and message passing should be *asynchronous*, *decoupled*, *evolvable*, using a *content-agnostic* communication channel.

Example

Consider an home automation solution that manages Air Conditioning (AC) systems. Three services compose the solution: *Sensor Reader*, *Data Receiver* and *AC Manager*. *Sensor Reader* is deployed inside the user's house. It is responsible for acquiring and forwarding temperature data. *Data Receiver* is a Web Server that receives temperature metrics and persists them in a database. *Data Receiver* is also able to provide aggregations over the data persisted in the database. *AC Manager* is responsible for managing AC units by evaluating the average temperature over the past 10 minutes, configuring an AC to generate cold or warm air. The three services must cooperate in providing a complete solution for automated AC management. The expected interaction between them is depicted in figure Figure 9.1 (p. 137).

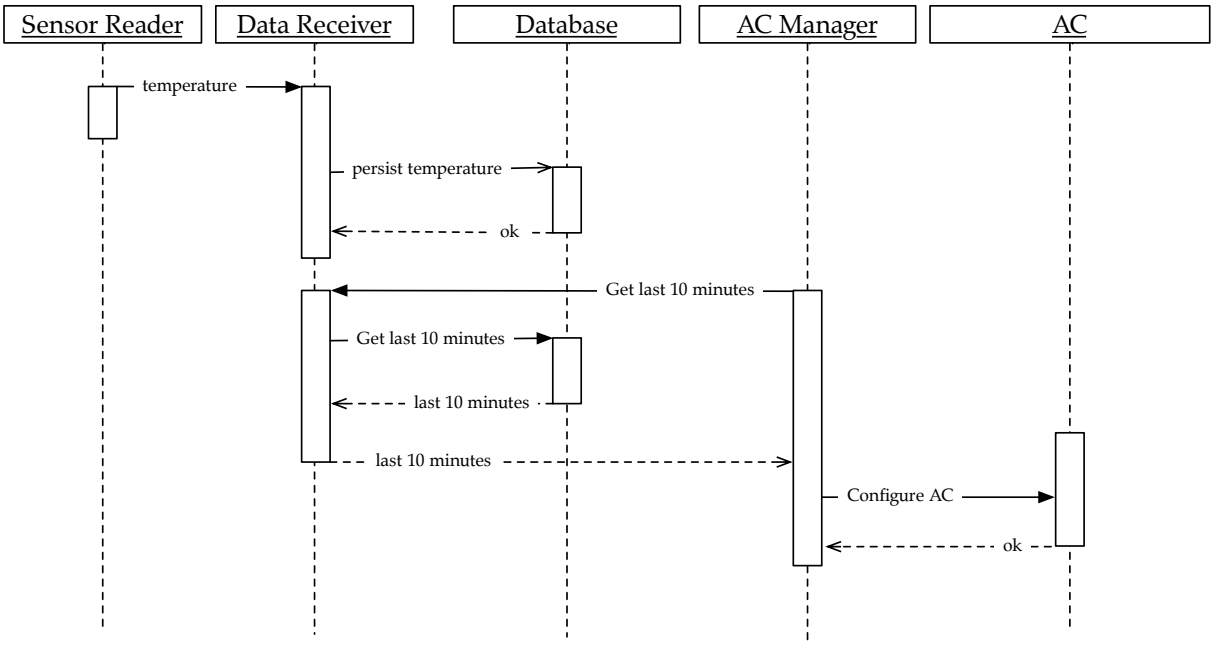


Figure 9.1: A microservice architecture-based system to capture and persist temperature metrics from a home environment, later used to configure an AC system. The arrows in the sequence diagram represent the massages exchanged between the components.

Problem

As the volume and complexity of interacting services increase, point-to-point communication channels become unmanageable, hindering fault-tolerance, resiliency, and scalability.

Services in a cloud application need to communicate with each other to cooperate. A typical communication strategy uses a client-server approach, limiting the communication to the two intervening service instances and requiring that the client knows how to connect

to the server, namely its hostname and server port. Cloud applications are deployed into dynamic hardware, which means that the internal server's addresses are not available during development time, rendering troublesome to use direct communication between services. Furthermore, when multiple instances of a service exist, the traffic needs to be balanced between all instances.

Considering the above, the need for abstraction over the communication between services is identified. Such a channel must enable passing any messages and correctly identify the sender and receiver of such messages. The communication channel must be scalable, ensuring that latency requirements are met even when handling large volumes of messages.

Forces

The following forces, represented in Figure 9.2 (p. 139), need to be balanced while considering the adoption of this pattern:

The following forces influence this pattern:

Decoupling. A sender does not need to know the network address of a receiving service to communicate with it.

Scalability. The communication channel needs to be itself scalable.

Resilience. Communication should be resilient, despite failures in the communication channel.

Persistency. Messages between services should be persisted until there is a confirmation that they have been processed.

Structure agnostic. The communication channel should be agnostic to the messages it orchestrates.

Dynamic and flexible. The topology of the system will evolve with time, with new services joining existing ones, and others leaving in real-time.

Payload security. The communication channel should support encrypted messages.

Channel security. The communication channel should be itself encrypted.

Latency. Introducing an indirection in communication increases the latency required for passing a message between two services.

Ordering and priority. The team needs to evaluate if message ordering and priority are relevant for implementation.

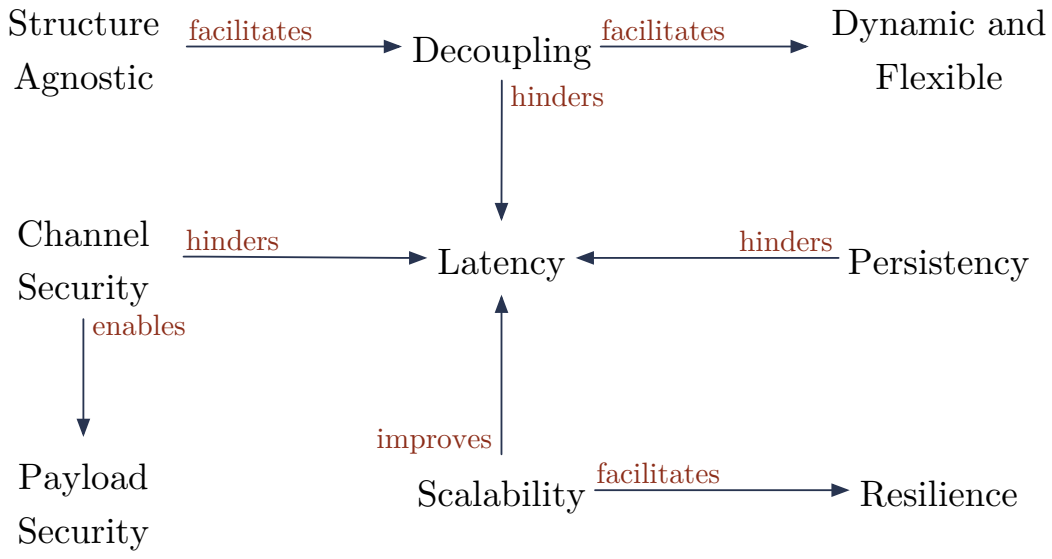


Figure 9.2: Relationship between the MESSAGING SYSTEM forces.

Solution

Use a MESSAGING SYSTEM*, colloquially known as* message queue*, to abstract service placement and orchestrate messages with the optimal routing strategy between them.*

A MESSAGING SYSTEM is responsible for routing messages between services, which can be both producers and consumers of messages. Messages can vary in size and contents, since the channel is agnostic of their internal structure, as long as they respect the adopted protocol.

MESSAGING SYSTEM works by creating one or more queues that work as a First in, first out (FIFO) data structure. Some implementations provide the possibility of prioritizing messages in the queue. Quality of Service (QoS) policies can also be applied, forcing consumers to confirm that they have successfully processed the message before it gets discarded from the queue. QoS ensures that a failing service will not remove a message from the queue without it being processed. If a service fails to acknowledge that the message has been processed in an acceptable period, the message becomes available for another consumer to process.

Most implementations support multiple message delivery strategies. RabbitMQ, which is one of the most adopted implementations, supports simple queues, exchanges with multiples queues, routing, topic-based consumption, and RPC [Piv07].

When implementing Remote Procedure Call (RPC), services can issue requests to the message queue and block waiting for an answer. A consumer would pick up the request, process it, and send it back to the queue, destined to the request sender. That first server would then receive his request and resume his computation.

Moving the responsibility of handling all communications to the Message Queue service makes it a single point of failure. For this reason, messaging services are typically deployed with redundancy, ensuring that communications between services will continue to work if some instances fail.

The concept of message passing systems has been available for several years, as middleware provides highly-observable communication strategies, namely one-to-many communication, providing dynamic connections among services. The initial reference to messaging applications as a means of communication between servers was first introduced on the 2001 patent *Message Queue Server System* [YH02]. More recently, several standards have been introduced, namely the *Advanced Message Queuing Protocol* (AMQP) and the *Message Queue Telemetry Transport* (MQTT) [Mag15].

Most implementations will enable the communication channel to use an encryption algorithm to protect the communication channel. Being agnostic to the message's contents, the payload itself can also be encrypted when needed, preventing data leaks even if the MESSAGING SYSTEM is compromised.

Ordering might or not be respected, depending on the adopted implementation. RabbitMQ, for example, can ensure processing order and even has support for priority and sharded queues [Rab20], with strategies other than a FIFO for delivering the messages.

Example Resolved

Considering the example described in section Section 9.2 (p. 137), the three services can communicate using a message queue based distribution in a message system, as shown in figure Figure 9.3 (p. 141). Message queues can be identified by a name and require consumers to subscribe to the queues from which they want to receive messages.

Initially, the *Data Receiver* service would subscribe to queues *metrics* and *requests*. *AC Manager* would subscribe to a queue named after it, *manager*.

Inside the house *Sensor Reader* would capture temperature metrics and send them to the message queue using the *metrics* queue. Asynchronously, *Data Receiver* would consume these messages and persist them in the database.

Periodically, *AC Manager* would require the last 10 minutes of temperature metrics to the message queue in the *requests* queue. *Data Receiver* would consume that message,

gather that information from the database, and sent it back to the message queue using the *manager* queue. Finally, *AC Manager* would consume those messages and configure the AC system with the appropriate behavior.

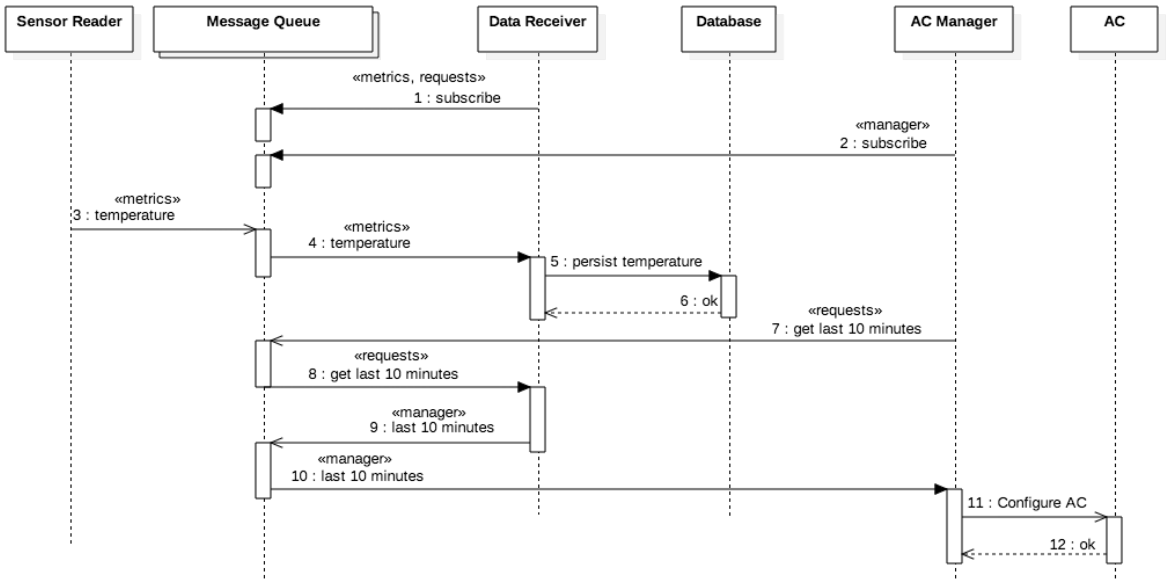


Figure 9.3: Communication between the three described services, routed via a messaging system. No two services communicate directly. Arrows represent the messages exchanged in the systems.

Resulting Context

In the context of engineering software for the cloud, message queues can abstract where services are located, eliminating the need for discovery mechanisms between them. Each service can communicate directly with one or more queues, requiring only the address of the Message Queue service.

Using message queues also facilitates service scaling. Services receiving traffic from outside channels should be scaled in order to handle the traffic. These would then inject messages in queues that are being consumed by other services. In such architecture, the queue's size can be used to understand if and how a service should be scaled, aiming at always keeping the message queue as small as possible.

This pattern can positively improve a cloud application as follows:

Decoupling. Using the MESSAGING SYSTEM to orchestrate messages between services further ensures their decoupling, facilitating the evolution of the application in a microservice architecture.

Scalability. Queues can be shared by multiple instances of a service, acting as a load balancer, rendering it trivial to scale a specific service.

Resilience. Messages can remain in the queue until its consumer marks them as processed. In case of failure, another instance can retry processing a message. The message queue software itself can also be deployed redundantly and have a disk persistence of its messages, to ensure resiliency.

Availability. While the MESSAGING SYSTEM becomes a single point of failure to the application, it can be deployed redundantly, ensuring its continuity if a node from it fails.

Security. Security can be improved by obscurity, as the services receiving messages do not need to be reachable from the message sending services. Also, the communication channel uses encryption to enforce a secure communication of all messages sent through it.

On the other hand, the following pitfalls are observable:

Complexity. , Increasing the level of indirection might make it harder to debug message passing in the application. For this reason, Facebook's Flux architecture, which is partially event-driven, explicitly disallows sending nested events.

Latency. Again, the indirection introduced will forcefully increase the message-passing latency when compared to point-to-point communication. Current message queue implementations, when co-located with both services and given the appropriate network conditions, can still ensure latency under 50 milliseconds [Ric].

Single point of failure. Without a redundant deployment, a failure in the messaging system will halt all interaction between the services.

Ordering and priority. The implementation can support ensuring ordering and priority if the team requires it.

Related Patterns

Message Queues are a more elaborate approach to Hohpe's Message Buses, which provided a basic communication channel between applications. In his book *Enterprise Integration Patterns*, additional communication patterns that most message queue implementations have adopted are described, such as PUBLISH-SUBSCRIBE CHANNEL or GUARANTEED DELIVERY [HW03].

Another version of the Publisher-Subscriber pattern was also documented by [BMR96].

This pattern introduces an approach to allow services to communicate without knowing their peers' location. This might not be acceptable at all times, mostly due to latency constraints. For those cases, SERVICE DISCOVERY [BCS15] can be applied.

A similar strategy described by the IO GATEKEEPER and related patterns in the telecommunication domain for managing the interaction between humans and systems [Han98].

SERVICE DISCOVERY can also be used to discover where the message queue is available in the infrastructure.

Messaging systems can be used to implement LOG AGGREGATION, by having services communicating their logs as messages, which are then aggregated by the log centralization service.

Known Uses

MESSAGING SYSTEM has a wide range of adoptions. At Conseil Européen pour la Recherche Nucléaire (CERN), it was used to make information available for multiple monitoring tools in multiple projects, namely in the Large Hadron Collider (LHC) [Cas+11]. A similar environment to the one presented in section Section 9.2 (p. 140) is described by Grgićm, along with details on how to instantiate it [GŠH16].

In another example, [Her+16] demonstrates how message queues can be adopted to acquire real-time data from trains and be used with Reactive Blocks[1] to facilitate collaboration in development and maintenance of software systems.

9.3 Service Discovery

In a dynamically allocated infrastructure, services require a discovery strategy to establish a communication channel. Abstract service network details by relying on an external mechanism that facilitates communication and balances traffic between two services.

[1] Project details available at www.bitreactive.com.

Context

Cloud applications are commonly composed of a multitude of services, which may be spread over multiple physical servers in different networks. In order for services to cooperate, they need to know how to communicate with each other, which implies the need for configuration or discovery of the hostname or IP and port where the required service can be reached. Furthermore, when a service has multiple instances, required in high availability setups, there might be a need to distribute traffic between existing instances evenly.

Example

An application server receives HTTP requests and queries a database server for information required to process the HTTP response. For scalability purposes, the database is distributed with multiple read replicas that vary in the number of instances considering the average system load. The service has no information about how the database servers can be reached due to the dynamic allocation of database instances. Figure Figure 9.4 (p. 144) represent a possible distribution of services among the existing servers of such a system.

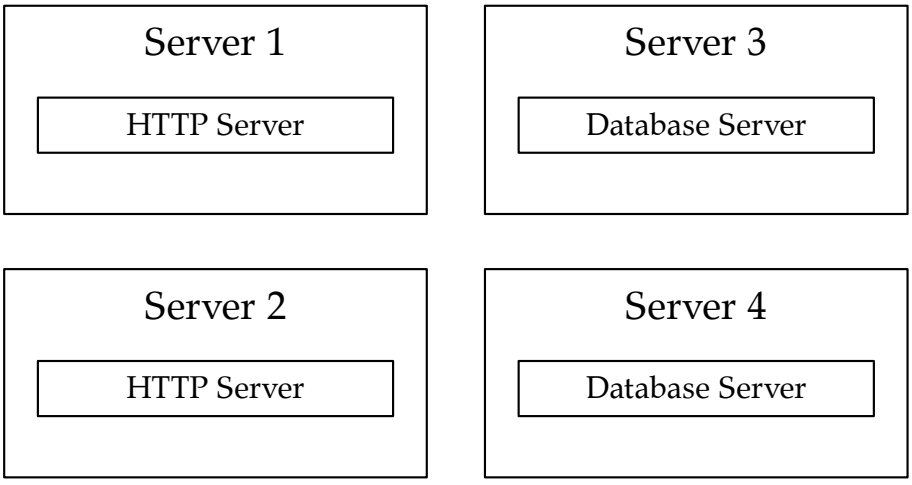


Figure 9.4: The four members of an infrastructure, each hosting a service.

Problem

In a dynamically allocated infrastructure, services require a discovery strategy to establish a communication channel.

Service decoupling is required as software gets deployed and scaled automatically in the cloud, enabling the scaling of individual software components when using dynamically

provisioned hardware. Deploying in these conditions leaves the client services unaware of where other services are allocated, requiring a discovery strategy to enable synchronous communication between them.

Forces

The following forces, represented in Figure 9.5 (p. 145), need to be balanced while considering the adoption of this pattern:

Real-time discovery. State must be updated when there is a change in the number of instances in a service.

Location decoupling. Services do not need to know where others are deployed to communicate with them.

Protocol agnostic. Work at the network level, supporting any protocol adopted by the services.

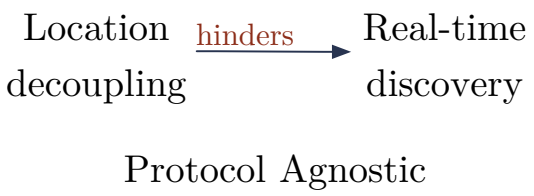


Figure 9.5: Relationship between SERVICE DISCOVERY forces.

Solution

Abstract service network details by relying on an external mechanism that facilitates communication and balances traffic between two services.

Use a new component to instruct a client service on how to reach the destination service. Implementations can vary from using a DNS server of a reverse proxy within each server.

The first approach consists of using a DNS service that is aware of the service deployment, creating one DNS entry per service, and keeping it up to date so that it will always resolve to the list of servers where the service is deployed. This approach requires forcing the deployed client services to use this DNS server.

The reverse proxy approach relies on deploying a proxy in each server. The proxy exposes a service port for each service and is aware of the deployment state so that it forwards each local port to where the service is actually deployed within the infrastructure.

Proxies work at the network level, which makes them protocol agnostic, seamlessly handling TPC, UDP, or HTTP.

Both strategies require that the proxy or DNS server be continuously aware of the deployment state. There are multiple strategies for doing so. One is to have a service registry where each service announces itself, along with dedicated software that periodically reads this information and updates the proxies. Another alternative is to query this information from an ORCHESTRATION MANAGER.

Both proxy and DNS servers can be configured on how to route traffic When multiple instances of a service are available, acting as a load balancer. The balancing algorithm might work, for example, by distributing the requests using a round-robin technique or in a smarter way, according to the target's resource availability.

Example Resolved

This technique requires an external orchestration mechanism to keep meta-information on the services running in the infrastructure, regarding hosts and ports. Each host machine has a proxy that periodically queries the orchestration manager and forwards a known local port to the host(s) and port of where a service available in the infrastructure. The applications expect a specific port to be available locally that will abstract the exact port and host where the service is running. Consider the example previously described: a web application is deployed with two HTTP Servers receiving external requests, which must communicate with one of the two other Database Servers to create a reply. For the HTTP servers to communicate with the database, they connect to the known local port instead of establishing a direct connection, leaving for the proxy to forward the request to an available Database server. Scalability is achieved by varying the number of Database or HTTP Servers independently, relying on the proxies on the HTTP side to properly identify available Database Servers and distribute the load between them, acting as an internal load balancer. This example is represented in Figure Figure 9.6 (p. 147).

Resulting Context

This pattern introduces the following benefits:

Real-time discovery. Changes to the infrastructure are immediately identified by the orchestration manager, which will reconfigure the proxies.

Location decoupling. Service development can ignore the actual physical location of other services it is integrating with, relying on the reverse proxy to forward traffic to the execution of the service.

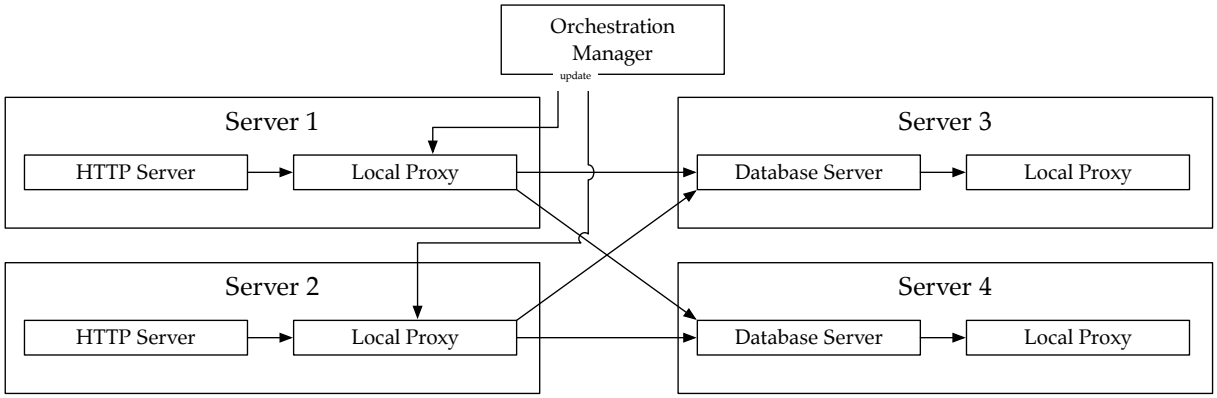


Figure 9.6: Proxy configuration example.

Protocol agnostic. Proxies work at the transport OSI layer or lower, hence, are protocol agnostic.

The pattern also introduces the following liabilities:

Monitoring. A mapping between a service and its running instances must be maintained at all time so that the reverse proxies are properly configured and only redirect traffic to active services.

Related Patterns

This pattern may be applied when CONTAINERIZATION is being used to isolate applications, facilitating communication between containers hosted in different servers, without requiring applications to integrate with discovery mechanisms individually. Information about service ports in each container can be injected using environment variables.

This pattern depends on an external mechanism that keeps track of each service in the infrastructure. An ORCHESTRATION MANAGER holds this information and could be queried for it.

Known Uses

A basic approach is presented by Wilder, keeping an Nginx reverse proxy updated according to meta-information extracted from running docker containers in the local machine [Wil15].

The reverse proxy Vulcanproxy [Com15b], together with the distributed key-value storage Etcd [Com15a] provides a reverse proxy service agnostic to the software using it. Integration with it requires each service to register itself with Etcd, or have an external service monitoring, which is less automated than the other solutions described.

A better solution is based on Apache Mesos [Fou15], which allows jobs to be spawned across multiple nodes, managing their allocation in Marathon, an infrastructure-wide init and control system for Mesos [Inc15a]. Using meta-information available with Marathon, a script can periodically update a proxy server on each machine in the infrastructure, forwarding a TCP or UDP port, named the service port, to the actual address where the application is running, despite it being local or in a remote machine [Wug15]. There are many implementations available to work with Marathon, including Bambo, an HAProxy auto-discovery and configuration tool for Marathon [Wug15]. There is also a script that can configure a local HA proxy, made available by Marathon's team [Inc15b].

Kubernetes has implemented this pattern by providing an embedded DNS server that automatically exposes all services deployed with it [Kub18a].

9.4 Summary

This section introduced two patterns for supporting service cooperation. MESSAGING SYSTEM introduces a message passing as a strategy to asynchronously exchange messages between services, while SERVICE DISCOVERY facilitates service discovery in a cluster, supporting synchronous interaction. Service discovery and communication are essential to enable service cooperation and vertical service scaling.

This chapter detailed the last two patterns from the pattern language introduced in Chapter 6 (p. 69). The following two chapters elaborate on how these patterns are being used in the industry.

Chapter 10

Industrial Case Study

We have elaborated on the intricacies of cloud development through literature research and technology experimentation. We proceeded to present a pattern language for designing cloud applications, enabling developers to make informed decisions for their cloud software design. We now question what the actual impact of the patterns in this language can be for developers. Are these patterns relevant? Are developers aware of the problems that benefit from the application of these patterns? Through the application of Semi-structured interview (SSI), we describe a case study with five local startup companies. The usage of startups in this study was motivated by the facilitated access to them, building cloud-centric products, and how much they depend on the efficient use of cloud computing. We evaluate how their design relates to the patterns that we have identified. As a secondary

objective, we expect our finding to provide valuable feedback to improve our patterns with new forces and implementation details.

10.1 Goals

The pattern language described in the previous chapter is supported by research and experimentation. Still, a pattern language is an ever-improving artifact. New information and realities can provide insight for improvement at any time. With this case study, we want to understand the correlation between a company's maturity and its level of adoption for the pattern language. We address Research Questions (RQs) 2, 4, and 5, by inquiring companies about their cloud problems and solution strategies, learning about their forces, and relating their pattern adoption with their company characteristics. During this process, we expected to identify additional details regarding the strategies applied to solve the identified recurrent problems, concretely new forces, and implementation details that can further improve the pattern language.

We hypothesize that there is a **a correlation between the maturity of a product or its company and the number of patterns they adopt**. We consider that a more mature product or company operates at a larger scale, with a larger team, and possibly with more complex operation requirements, such as geographic distribution. We consider a product or company to be more mature than others if it has a bigger team or operates on a larger scale.

10.2 Methodology

The case study works with five companies building cloud products and evaluates their cloud practices. These companies share a similar business stage (not product stage), as they were all past the seed stage and undergoing client expansion, with most already having secured their first international clients.

For this case study, we aim to understand how each of these companies approached cloud development. We had limited access time to each company, which discarded observational methods. As such, this research was designed using SSIs [Ada15]. SSIs provide a framework for capturing qualitative data by following a script characterized by its open questions. The interview protocol is described in Section 10.3 (p. 151). SSI ensures the interviews capture detailed qualitative information while providing the possibility to probe the respondent for additional information concerning his system, allowing him to digress over details that might have been disregarded in a structured interview. The face

to face interaction motivates respondents to be more invested in the study, providing additional details about their systems that they would likely not provide in a closed survey with a less interactive approach, such as using questionnaires [LA94].

Interviewers can use probing during a SSI as a technique to stimulate the respondent to clarify or elaborate on a topic. An example would be to ask "Can you please elaborate on that?" on an incomplete response or echo the response so that the respondent evaluates if he has missed any details [HB09]. It is up to the interviewer to understand when and how to use probes during a SSI [HB09].

Lazarsfeld states that the SSI interviewer should be a domain expert, as much of the dialogue might use ambiguous words [Laz54]. Barriball claims that an expert interviewer can obtain more and better information out of an interview [LA94]. To ensure consistency and that an expert holds the interviews, the author was responsible for performing the five interviews.

On the downside, SSIs are time-consuming, given the need to physically conduct the interview and then process all the unstructured data captured from it [Ada15]. As such, SSIs may need other supporting methods to achieve statistically relevant results. Recording the session enables the interviewer to be focused on the respondent and his answers while preventing the loss of valuable information from the session [CC06].

10.3 Interview Protocol

The following sections detail the interview script. Questions are organized into the categories: Introduction (I), Infrastructure Management (IM), Orchestration and Supervision (OS), Monitoring (M), Discovery and Communication (DC), and an Hypothetical Scenario (HS).

10.3.1 Introduction

The interview begins by contextualizing the respondent about the process and goals for the interview. The respondent is made aware that the conversation is informal and semi-structured, motivating a deviation from our questions to further elaborate on any relevant topics. We also made evident our goal to understand the team's cloud software practices and related business intricacies. We ask the following questions during this phase of the interview:

I1. Briefly describe your business and product.

Different businesses demand different cloud strategies. Specific requirements, such

as dynamic hardware scaling or high-frequency deployments range from irrelevant to mandatory for the business to thrive. Asking the interviewees about their business enables understanding the requirements and expectations set for the product.

I2. Who is the typical user of your software? What value is being added to him?

This question improves on the previous one, surveying the respondent about the relevance of his product, enabling us to familiarize ourselves with their operations.

10.3.2 Infrastructure Management

Cloud orchestration primes for its automation capabilities, which can be implemented using INFRASTRUCTURE AS CODE or AUTOMATED SCALABILITY. This part of the interview inquires about deployment strategies adopted. We ask the following questions during this phase:

IM1. Would you describe your architecture as a monolith or a service-oriented architecture? Has it always been that way? What made you design it as it is?

Cloud applications typically start as monoliths [Fow15], given the development agility and simplified operations that these require [Sti15; Bon16; Ric17b]. Service Oriented Architecture (SOA), despite more complicated to operate, facilitate scaling in the cloud. Most teams either adopt SOA from scratch or eventually refactor their applications into using it. Understanding the respondent's application architecture allows us to infer their cloud approach strategy and vision.

IM2. Can you draw and describe your architecture? Can you identify the critical components from your architecture?

An architectural draft identifies what services compose the product and how they interact with each other. This provides a shared vocabulary to support the remaining interview.

IM3. Can you describe how many users and traffic volumes you are managing right now? If you have multiple deployments, consider the largest for this question.

Active daily users and traffic volumes are critical metrics for adjusting the scale at which an application needs to perform. Companies managing smaller volumes might not have yet felt the need to scale their application, while large companies are seasoned at handling scale variations.

IM4. What cloud infrastructure do you use? Which services are they providing you?

This question probes what cloud providers and external services are adopted, as well as how they are configured and coordinated. Services are typically made available in the form of metal, platform, infrastructure or software as a service. Different levels require different investments in automation and orchestration management.

IM5. How many different instances of your application are you currently managing? For what purposes? How are you managing them?

While some cloud applications are multi-tenant, others need to be deployed on a per-client basis. Single-tenant applications have further isolation, which might be a requirement from the product. Independent deployments per client require increased operations effort, as more application instances introduce more possible points of failure.

IM6. Can you describe your deployment strategy?

A manual deployment strategy is error-prone and demanding on human resources, limiting the operations' efficiency. By probing the respondent about their strategy, we expect them to acknowledge this discuss their strategy and how their level of automation influences operational errors and costs. We expect to observe that less mature companies have less automation and more frequent errors during deployments. In contrast, more mature companies use automation to make their operations reproducible and less error-prone. Possibly related patterns: AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE.

IM7. Is there any deployment automation in place?

This question is aims at acquiring further deployment details that relate to automation if such has not been presented in the previous answer. Possibly related patterns: AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE.

IM8. Can you describe any recurrent deployment issues?

Software development and orchestration of growing application will eventually face issues. The way developers handle those issues, the frequency at which they happen, and how comfortable the team is with coping with them on a daily basis might define when a team stops further automating their operations. We want to understand the respondents' history with deployment issues, their impact, and how the team addresses them. Possibly related patterns: AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE.

10.3.3 Orchestration and Supervision

Large scale software projects quickly require complex clusters composed of a multitude of servers and services. Has the infrastructure grows, the team must assume the responsibility of operating a more complex infrastructure. This part of the interview explores the level of maturity for the interviewee's operations, from the moment the software is packaged to its daily orchestration. The following questions are asked during this phase of the interview:

OS1. How do you package your software?

Software development usually takes advantage of reusable pieces of other software. As such, it typically requires a multitude of dependencies to execute, from libraries, configurations, and the binaries themselves. All these items need to be present and with the correct versions for the software to execute properly. Some strategies for managing software dependencies are worst than others. For example, relying on the operating system's package manager to make a library available might result in a wrong version installed at a given point in time. Packaging software as a Zip file with all its dependencies would require a person or process to follow a procedure to copy the package, install its dependencies, and sets up the application so that it is correctly installed. Updating from a Zip file also requires a detailed protocol, as some files might not be overwritten, while others might have to be deleted, preventing a simple extraction of the Zip file. Using container technology ensures that all dependencies are available using the proper versions and that the software deployment is facilitated and configured via environment variables. This question probes the respondent about their strategy to package software, capturing the requirements and limitations that led to the adoption of this strategy. Possibly related patterns: CONTAINERIZATION.

OS2. How do you deploy your software packages?

Following the previous question, we want to understand what strategies the respondent adopts to move his software from development to production. As seen in previous chapters, a manual deployment operation is error-prone and time costly. We expect less mature companies to use manual deployments and occasionally to often struggle with deployments. At the same time, more mature companies would use further automation, with a reduced failure rate and deployment time. Possibly related patterns: CONTAINERIZATION, ORCHESTRATION MANAGER.

OS3. Can you describe the process of setting up a new instance of your cloud application?

In the context of the cloud, it is necessary to allocate hardware before deploying the software. Several criteria influence how often new systems are deployed, either for development, staging, or production. This question probes the strategy for setting up new product instances. Companies with small teams and single-tenant applications will not have to do this often. They might be comfortable with manually allocating the hardware required to run the new system using their cloud provider web interface sporadically. Multi-tenant or more mature companies will want to automate this process as part of their deployment automation, ensuring that the allocation of cloud resources is just another step in their deployment pipeline, again, reducing time and error-proneness. Possibly related patterns: AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE, ORCHESTRATION MANAGER.

OS4. Can you describe the process to update the application or part of it?

Previous questions have addressed the strategies to package the software, set up a new environment, and move software onto it. That infrastructure and software will not be static, but evolve with the product, as new requirements become features and the product expands. We want to understand how the responded handles their infrastructure evolution and software updates. Again, we expect to see less mature companies updating the infrastructure manually via their cloud provider web interface, as well as needing to introduce relevant downtime for updating the system. On the other hand, we expect more mature companies to automate their operations at the infrastructure and application level, requiring little to no downtime to update. Possibly related patterns: AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE, ORCHESTRATION MANAGER.

OS5. Do you have any automation for identifying and recover failing software?

It is a generally accepted fact that software fails. Here we probe the respondent about his strategy to stay aware of his software status and if he has any strategy for automatic software recovery. Failure recovery automation can reduce system downtime on failures, preventing the degradation of confidence from the users in the product. Possibly related patterns: ORCHESTRATION MANAGER, AUTOMATED RECOVERY.

OS6. Considering your recovery automation strategies, how do you ensure these strategies are working?

Automated recovery strategies can significantly reduce system downtime. Nevertheless, if these are not exercised frequently in a controlled way, they may

be themselves faulty, without the team being aware of it. We want to learn if the respondent has any strategy to verify his recovery mechanisms, making sure that the system will indeed recover in the event of a failure. Possibly related patterns: FAILURE INJECTION.

OS7. Do you have scheduled jobs running in the system? If so, how are they triggered, and where are they executed in the infrastructure?
Scheduled jobs are a ubiquitous requirement for cloud applications. Examples range from running periodic backups to sending transactional emails. These can be challenging to implement when the application begins to scale horizontally. Where should the job execute? How to ensure it executes in the proper machine(s)? How to implement redundancy to ensure jobs will still run even in case of failing machines? These are just some of the recurrent problems that might appear. We prove how the respondent addresses job scheduling, as well as which issues he often has with their implementation. Possibly related patterns: ORCHESTRATION MANAGER, JOB SCHEDULER.

10.3.4 Monitoring

Software is prone to failure, both from internal (e.g. faulty code) and external reasons (e.g. faulty hardware). Quickly identifying failures is paramount for an also quick response. This part of the interview probed the respondent about his strategies to identify failures in his application. We ask the following questions:

M1. Do you have continuous monitoring for the application? If the whole infrastructure suddenly fails, would you still be notified of the issue?
Monitoring is essential to reduce the team's reaction time to issues that are not automatically recovered. In such scenarios, monitoring from within the infrastructure might provide a biased understanding of the actual application's state, reporting a functioning system that. in fact, is not. Here we probe the monitoring strategies from the respondent to his system, evaluating their capacity to prevent basic false positives. Possibly related patterns: ORCHESTRATION MANAGER, AUTOMATED RECOVERY, EXTERNAL MONITOR.

M2. Do you store your log files? How and for how long?
The value of information is often under-appreciated, which can motivate its premature destruction for diminishing cost savings. We want to understand how these logs are stored and accessed. Keeping the logs close to their source can result

in complex debugging scenarios, such as a team member having to access several machines to gather data deemed relevant. We seek to understand if the respondent has ever gone through a situation were relevant system information was unavailable for this scenario and how easy it was for the team to explore log data from multiple sources. Possibly related patterns: LOG AGGREGATION.

M3. What is your common application log level of verbosity? Have you ever decided to increase the level of verbosity to ensure you capture valuable information? Why and for long?

Decreasing log verbosity levels is common when the system is stable or even right before the first deployment. Unexpected situations come unannounced, and a reduced logging level can result in the developers' loss of relevant debug information. This question probes the respondent about their strategy to maintain a balanced log verbosity level. Possibly related patterns: PREEMPTIVE LOGGING.

M4. Consider that the system is down. What would be the typical process to diagnose the problem?

This question probes the respondents about their protocol for addressing failures. We want to understand if there are clear protocols and strategies to identify and address potential issues in the system. Possibly related patterns: LOG AGGREGATION, PREEMPTIVE LOGGING.

M5. What would you need to do to leverage logs from multiple sources to debug an issue?

Relevant information to debug an issue is typically spread between multiple servers and services. Leveraging data from multiple sources might not be trivial. This question probed the respondent about their protocol for addressing failures, evaluating how easy it is to leverage logs disperse across multiple machines. Possibly related patterns: LOG AGGREGATION, PREEMPTIVE LOGGING.

10.3.5 Discovery and Communication

Modern cloud software scales horizontally. While doing so, services require strategies to discover and communicate in order to cooperate. We probe the strategies in place in the respondent's cloud application to enable cooperation between services. The following questions are asked during this phase of the interview:

DC1. Do you dynamically scale your system? How often and how?

Dynamically scaling the system is essential for any cloud application to remain

cost-efficient. It requires careful consideration regarding architecture, automation, and monitoring, to name a few. We want to understand what architecture considerations enable the product to scale seamlessly. Also, what strategies are adopted to decide when and how to scale. Finally, we want to understand the limitations of those strategies, how confident the respondent is on their execution, and any frequent issues that result from them. Possibly related patterns: AUTOMATED SCALABILITY, ORCHESTRATION MANAGER.

DC2. What technologies do you use for facilitating communication between your services, either synchronously or asynchronously?
When applications scale, instances, or application components need to cooperate, despite being allocated in dynamically allocated hardware. As such, the team must adopt strategies for the services to find and communicate with each other, being it synchronously or asynchronously. This question follows on the previous one by probing the strategies adopted to facilitate cooperation between services. Possibly related patterns: SERVICE DISCOVERY, MESSAGING SYSTEM.

10.3.6 Hypothetical Scenario

At this stage, the respondent is presented with the following hypothetical scenario:

HS1. Imagine that you have to cope with a tenfold traffic increase. I want to revisit your previous responses and evaluate what you would like to do differently.
The respondent must revisit his previous responses, discussing what would need to change and how. The question is adapted per interview to ensure that a relevant challenge is proposed to the respondent. Providing an extreme scaling scenario ensures the respondent forces himself to consider all the intricacies of scaling his application.

10.4 Case Study: LabOrders

LabOrders[1] is a marketplace where research laboratories can buy their equipment and consumables. Sellers use the platform to expose their products to laboratory managers, which can manage their purchases and laboratory budget. The interview with LabOrders took place on October 18th, 2018, with the CEO, Tiago Carvalho.

[1] Learn more about LabOrders at https://home.laborders.com/.

10.4.1 Product Overview

LabOrders started their business to provide a unique marketplace for every material a laboratory could require. Later on, they have grown into a full-fledged laboratory management tool. At the time of the interview, LabOrders facilitates managing projects and their budgets, grant reimbursement from financing institutions, and human resources.

From the laboratory side, there are two user personas: the researcher, which requires material for his work, and the laboratory administrator, which manages the laboratories, e.g., by approving the researcher's orders.

Currently, LabOrders has over 3000 active monthly users and is ISO 27001 certified, which ensures their information security practices [Int]. This certification required extensive investment in infrastructure security. One of the requirements was to protect the application behind a VPN server that runs on a separate server in the same infrastructure.

LabOrders architecture diagram, as provided by the team, is depicted in Figure 10.1 (p. 159).

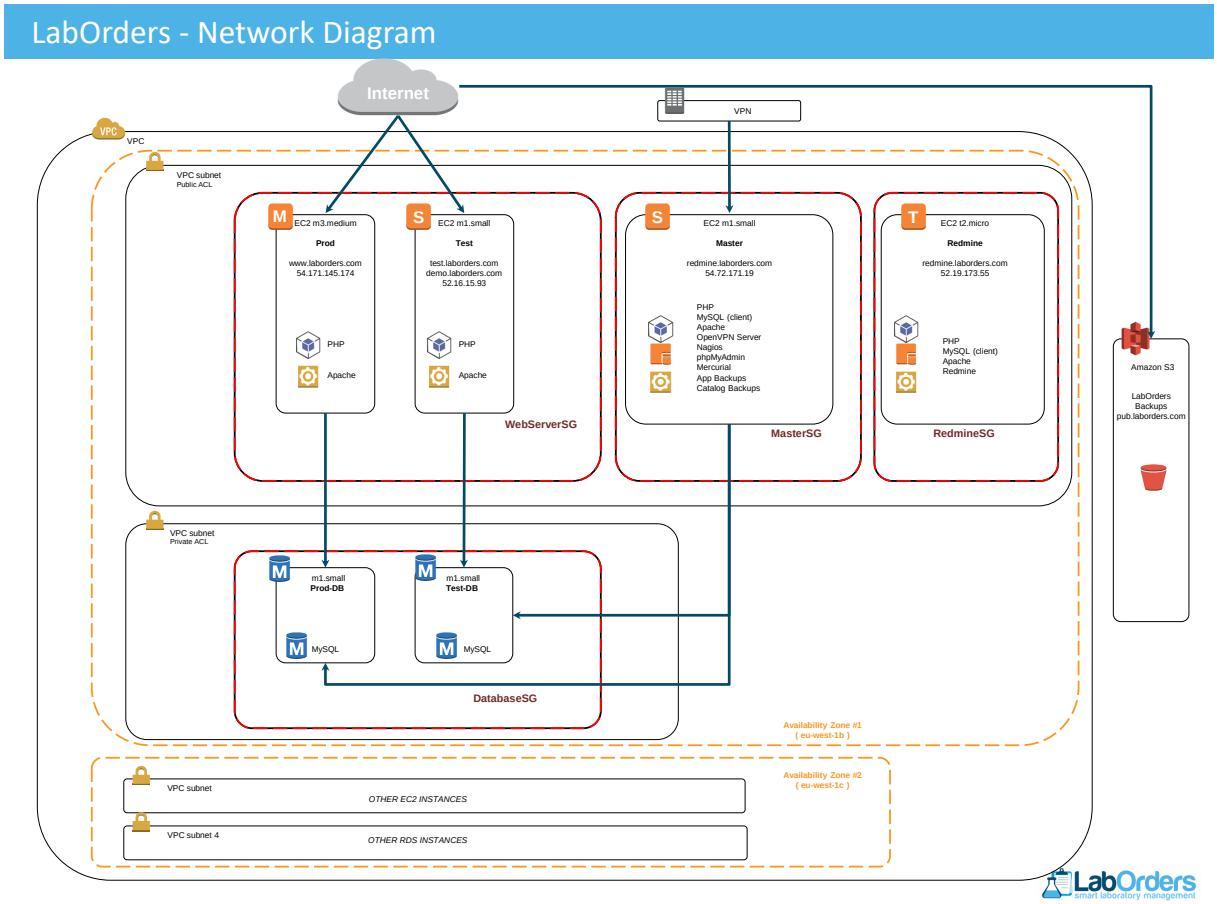


Figure 10.1: A graphical representation of the architecture for LabOrders, provided by the respondent.

At the time of the interview, LabOrders application was a monolith running on a single server provided by Amazon Web Services (AWS) Elastic Compute Cloud (EC2). There were external service dependencies, such as Sphinx[2], which executed in the same server, and a Structured Query Language (SQL) database provided by AWS Relation Database Server (RDS).

Every three months, the team did a capacity review and estimation, manually adjusting the infrastructure size to the upcoming trimester. By the time of the interview, there had never been the need to allocate a second server, but instead, the existing server scaled vertically, that is, by increasing the EC2 instance capacity (e.g., Disk Space, Central Processing Unit (CPU), Memory, etc) that would best adapt to their expected requirements.

10.4.2 Infrastructure Management

The infrastructure was allocated using the AWS web interface, and there was rarely the need to change it. Dependencies installation and setup were automated using INFRASTRUCTURE AS CODE with proprietary bash scripts that can Secure Shell (SSH) into an EC2 instance and set it up to receive the application. The pattern was only partially implemented since there was no automation for setting up the infrastructure itself. The deployment process was described in the team's wiki and performed manually every time.

The respondent considered the deployment and migration process troublesome, given that it is slow, requires downtime, and is error-prone. The average time for deploying the system was of one hour, during which the system was offline — having the system offline for such time required updates outside office hours, typically during the night and on weekends. When the update process failed, there was no automated rollback strategy; the team needed to recover a database backup and upload a previous code version to get the application back up.

Despite disliking the complexity of the deployment process, LabOrders was yet to explore new strategies as there had been little issues with their approach. The team was not considering adopting automated deployments as they felt it would require a substantial time investment. They were still motivated to explore additional automation to increase the frequency at which they could update the application. At the time, they deployed on average once every three weeks.

[2] Sphinx is an open-source search engine. Learn more at http://sphinxsearch.com/.

Mercurial provided source code control version. The deployment process used the production server as a Mercurial remote, with the team pushing the code to that remote when they want to update the system. This action replaced the code in execution, which acted as an update strategy. The team handled migrations manually.

10.4.3 Orchestration and Supervision

The application required communication to client's Enterprise Resource Planning (ERP) systems, protected via VPN connections. The production server kept an open VPN connection for each client's infrastructure. A Cron job frequently verified if the VPN connections were active and restarted them if they dropped. Cron also triggered sending emails to clients and the daily backups, which created a local data copy and uploaded another to Simple Storage Service (S3).

10.4.4 Discovery and Communication

The application was deployed in a single server and connected to a single database with a static address. Given that all components were statically allocated, the team believed that there were no reasons to adopt a discovery strategy.

10.4.5 Monitoring

The EC2 server hosting the Virtual Private Network (VPN) server also executed a Nagios[3] server. Nagios is a monitoring tool which was capturing metrics from the production server, application logs from the webserver and application, and issued specific HTTP requests and verified their responses for correct behavior. In case of failure or high resource usage, Nagios generated an alert for the team.

We can observe that deploying Nagios in the same Virtual Private Cloud (VPC) to where the application was deployed did not implement EXTERNAL MONITOR, since it could result in false positives regarding the system's availability, as described in the pattern description. A false positive could happen if the VPC blocked internet traffic but allowed internal traffic.

Application log files were kept in the production server and backed up to S3 as part of the daily backups. The application stored log files for three months.

[3] From their website, Nagios provides enterprise-class Open Source IT monitoring, network monitoring, server, and application monitoring. Learn more at https://www.nagios.org/.

The respondent described that they often adjust their level of PREEMPTIVE LOGGING so that relevant debug information is available for the team to debug future issues.

LabOrders application had its proprietary implementation of LOG AGGREGATION, given that the application's web interface for administrators exposed a log viewing interface. This implementation was limited, though, becoming unavailable if the infrastructure went offline. Furthermore, the design only supported log files from one server, and it would have to be rethought to cope with multiple servers.

10.4.6 Summary

Company name	**LabOrders**
Company size	1 – 10
Active monthly users	1k – 10k
Product operation strategy	Single system, limited users
Pattern name	**Adoption**
INFRASTRUCTURE AS CODE	○
AUTOMATED SCALABILITY	
CONTAINERIZATION	◐
ORCHESTRATION MANAGER	◐
AUTOMATED RECOVERY	
JOB SCHEDULER	
FAILURE INJECTION	
PREEMPTIVE LOGGING	●
LOG AGGREGATION	○
EXTERNAL MONITOR	○
MESSAGING SYSTEM	
SERVICE DISCOVERY	
Adopted patterns count	1

Table 10.1: Overview of the pattern language adoption by LabOrders. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented.

Table 10.1 (p. 162) resumes the pattern adoption by LabOrders as gathered from this interview.

The respondent identified that their design undermined scaling the system, recognizing the need to change it in the future. The first change would move the system state from being stored in files that reside in the server to external file storage, specifically an AWS

S3 bucket. After the state had been moved away from the machines, it would become possible to add additional machines and distribute traffic between them using a load balancer. INFRASTRUCTURE AS CODE would have to be further implemented also to orchestrate the infrastructure.

While scaling the monolith horizontally would enable scaling the system as a whole, it would not allow for the most efficient resource allocation, as well as it would continue to grow to a size that would be complex to develop. For optimizing infrastructure allocation and facilitating scaling development, the respondent also considered that they needed to redesign their monolith as microservices.

The update process was considered troublesome and slow. The team was evaluating the possibility of always creating a new packaged environment for deploying the application using Docker and CONTAINERIZATION, which could later enable the adoption of an ORCHESTRATION MANAGER.

When asked about the possibility of handling the system at a larger scale, the respondent considered the need for proper LOG AGGREGATION, so that it became trivial to access log files from any machine and application instance in a single place, opposed to having to access the server and its logs manually, which was how they implemented PREEMPTIVE LOGGING. While they had monitoring over their application, it was implemented with a Nagios server that shared the same infrastructure and could have a biased observation of the system's state. EXTERNAL MONITOR would provide a more reliable observation of the system's state.

Finally, when considering an increased scale and more frequent and automated deploys, the respondent identified the need to create an automated test pipeline to increase his confidence level.

10.5 Case Study: HUUB

HUUB[4] manages shipping logistics for e-commerce businesses, mainly focused on the fashion industry.

The product manages the life cycle of their client's supply chain. Clients send products to their logistic centers, which stores and then forwards the goods to the customer. HUUB handles the purchase order and guarantees that the product reaches the customer in a timely fashion. The client has visibility of the individual tracking history of any product through their platform.

[4] Learn more about HUUB at https://www.thehuub.co/.

HUUB operates out of two logistics centers, one in Portugal and another in the Netherlands, managing about 300 daily shipments and rapidly growing. The interview with HUUB happened on October 23rd, 2018, with senior engineer Luís Melo.

10.5.1 Product Overview

HUUB's platform core service, named Spoke, started as a monolith, but recent work has decoupled its functionalities into microservices. Moving to microservices was motivated by the need to adopt a more agile development framework than the one in which the product was initially developed, as well as adopting newer technologies.

Initial decoupling served as a testbench for multiple technologies, resulting in different services implemented with different frameworks. The original monolith, Spoke, was implemented with the *web2py* Python web framework. The first services decoupled were STEM and Quality Check, implemented using Laravel and Django, respectively. The team adopted Django as the preferred framework and used it for all subsequent services.

At the time of the interview, several other services had already been decoupled. HUUB was running the following services:

Spoke. The user interface for all account managers and clients to interact with the product.

STEM. The shipping management system, responsible for overseeing the inbound and outbound product shipping.

Quality check. Supported the manual process of checking product quality from incoming batches, preventing faulty products from being shipped to final customers.

3PL. Integration with third-party logistics systems, enabled the creation of transport requests with partner shipping companies.

Tracking. A redesign implementation of the tracking service. This service has been decoupled from STEM.

Email. Send emails to clients regarding their shipped parcels. This service has also been decoupled from STEM.

10.5.2 Infrastructure Management

HUUB adopted AWS as cloud provider. At the time, there was no automation implemented for setting up the infrastructure. Infrastructural changes were manually performed using the AWS web interface by a team member, as needed.

There were two deployments environments of the whole application: one for production and another for Quality Assurance (QA).

While infrastructure allocation was a manual process, deployments were automated using a Jenkins pipeline. For the QA environment, commits to Git's master branch triggered the deployment pipeline, deploying the latest version of the application into the QA environment automatically.

Automated deployments used INFRASTRUCTURE AS CODE by adopting AWS CodeDeploy[5] service to deploy the application on the already allocated infrastructure.

The production environment used the same strategy, but the deployment pipeline was triggered manually instead of automatically. Such enabled the team to control when and what is deployed to the production system.

Each service had its deployment pipeline with a CodeDeploy configuration. Configurations included the details required to establish the connection to the other services.

CodeDeploy was configured to pull the latest source code version directly from the git repository, set up the necessary dependencies, and deploy it. No packaging or software isolation strategy was adopted.

A limitation of this approach was that there was no maintenance in the hosting environment, which could clutter the servers. Previously installed dependencies remained in the machine, even after they were no longer necessary. Eventually, there would be the need for manual intervention to either remove unwanted or conflicting software from the server or delete those servers and create new, for a fresh start.

A recurrent deployment issue with the production environment happened due to not having the production service configurations versioned with the source code, for security reasons. This option resulted in recurrent deployments with erroneous configurations, which introduced runtime errors that required manual intervention.

Some services deployment was not yet automated using Jenkins and CodeDeploy, requiring manual deployment to both the QA and production environments. The respondent described an episode when he had to deploy one of these services but failed to

[5] CodeDeploy is an AWS service for setting up infrastructure and deploying software onto it. Learn more at https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html.

provide it with the required configurations, which resulted in downtime for that service during the time required to acquire the proper configuration from his colleagues.

The team wanted to explore Terraform[6] for further increasing their adoption of INFRASTRUCTURE AS CODE, enabling automating the orchestration of AWS services programmatically as well. Doing so would provide ground for implemented AUTOMATED SCALABILITY. Furthermore, using INFRASTRUCTURE AS CODE would further formalize knowledge about the infrastructure using the source code, which could act as a reference for new collaborators.

10.5.3 Orchestration and Supervision

To retrieve data from remote integrations, Jenkins periodically triggered jobs that performed tasks in the application's components, from databases to servers. Since the hardware was static, these jobs had a static configuration, describing to which servers they should connect and their behavior. This configuration was a simplified implementation of a JOB SCHEDULER, without the support for running the jobs in dynamically allocated hardware.

The team had no strategy implemented for AUTOMATED RECOVERY and had never considered adopting an ORCHESTRATION MANAGER.

10.5.4 Discovery and Communication

Kafka, a MESSAGING SYSTEM, was used to queue asynchronous tasks such as sending emails or process product tracking events. For synchronous communication, each service had a bundled configuration file describing the connection details for any remote services needed. Communication happened via HTTP and Representational State Transfer (REST) APIs.

The team had not implemented SERVICE DISCOVERY, but the respondent considered adopting load balancers as a strategy for SERVICE DISCOVERY if, or when, he has to address dynamically scaling the system.

10.5.5 Monitoring

The initial effort towards monitoring the system was pursued by having the team capture server metrics using AWS CloudWatch. Captured CPU metrics from Spoke servers

[6] Terrafom facilitates creating infrastructure as code operations. Learn more at https://www.terraform.io/.

generated an alarm sent to the team whenever their load was consistently above 80%. Jenkins was also used to monitor the application by having a pipeline triggered every thirty seconds, which made an HTTP request to an endpoint in each service. If the service failed to respond, the team would become aware that there might have been an issue with the service. This Jenkins instance was deployed in the same VPC as the application. As such, this monitoring strategy does not qualify as a EXTERNAL MONITOR.

PREEMPTIVE LOGGING was considered, motivated by the multiple times the team lacked relevant data for debugging issues observed in production. Third-party integrations, such as having an e-commerce system notifying HUUB about new purchase orders, generated most of these production issues. The respondent identified the need to further improve their logging strategy by logging all incoming and outgoing requests to provide a detailed history of all interaction with third-party services.

LOG AGGREGATION was partially adopted as each service also writes its log files to its dedicated database. While there was not a single place where the team could mix and match log files despite their source, they used a SQL client to connect to all relevant databases and from a single client visualize log files from multiple sources. The team experimented with Graylog for LOG AGGREGATION but had not had the time to implement integration with it yet.

10.5.6 Summary

Table 10.2 (p. 168) summarizes the pattern adoption at HUUB. INFRASTRUCTURE AS CODE and MESSAGING SYSTEM were fully implemented, with the team already experimenting with LOG AGGREGATION and JOB SCHEDULER, and a roadmap for implementing AUTOMATED SCALABILITY and PREEMPTIVE LOGGING. Probably related to their intention to implement more patterns, HUUB was well aware of the limitations they had at managing their infrastructure. The team was strongly committed to automating infrastructure management programmatically. By doing so, HUUB would enable AUTOMATED SCALABILITY, which would unlock new challenges such as the need to route traffic across multiple instances of synchronous services or optimizing service placement by having a single machine hosting multiple services.

Company name	HUUB
Company size	10 – 100
Active monthly users	1 – 100
Product operation strategy	Single system, limited users
Pattern name	**Adoption**
INFRASTRUCTURE AS CODE	●
AUTOMATED SCALABILITY	◐
CONTAINERIZATION	
ORCHESTRATION MANAGER	
AUTOMATED RECOVERY	
JOB SCHEDULER	○
FAILURE INJECTION	
PREEMPTIVE LOGGING	◐
LOG AGGREGATION	○
EXTERNAL MONITOR	
MESSAGING SYSTEM	●
SERVICE DISCOVERY	
Adopted patterns count	2

Table 10.2: Overview of the pattern language adoption by HUUB. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented.

10.6 Case Study: Infraspeak

Infraspeak[7] develops software as a service for facility managers and technical assistance. The software facilitates preventive and corrective maintenance, identifying where maintenance is going to be needed before failures happen. Furthermore, it manages the intervention process of failures. Their clients range from hotels, facility management companies, or retail. Infraspeak has clients in Portugal, Angola, Mozambique, Cape Verde, Gibraltar, and Brazil.

The interview with Infraspeak took place on October 17th, 2018, with the CTO of the company, Luís Martins.

[7] Learn more about Infraspeak at http://home.infraspeak.com/.

10.6.1 Product Overview

Infraspeak's product supports maintenance businesses by tracking every maintainable asset and providing a communication channel for tracking maintenance operations between the stakeholders. The stakeholders are:

The customer. Reports issues with his assets through Infraspeak Direct;

The client. Pays for using the product and uses it to oversee all managed assets and their state

The technicians. Employees from the client that have a mobile application to acquire and update information from managed assets and their interventions.

Infraspeak designed its product as a monolith and has not yet felt the need to move to microservices. The initial version was served out of a single machine, using a typical Linux, Apache, MySQL and PHP (LAMP)[8] stack. At the time, Infraspeak was managing 800 000 tasks per month, translating to approximately 600 requests per second.

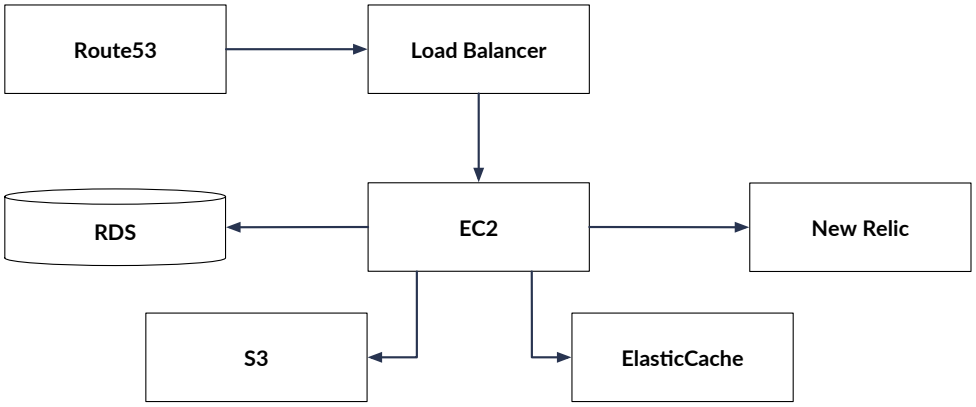


Figure 10.2: A graphical representation of the architecture for Infraspeak.

Figure 10.2 (p. 169) shows the architecture for Infraspeak. Given the need to preemptively prepare for horizontal scaling, the team migrated state to AWS Services, namely by adopting AWS RDS for the database, AWS S3 for file storage and ElastiCache for caching. At the time of the interview, there was no redundancy on service allocation, which meant that a failure in any service would have a critical impact on the system.

When confronted with the need to scale the system, the respondent considered that the initial scaling strategy would continue with a single-server system, but create one replica for each geographic region, for example, have one for Portugal and another for Brazil. A second scaling tier would change the EC2 instances, first by increasing them

[8] LAMP, or Linux, Apache, MySQL, and PHP is a standard software stack for web development.

vertically and then horizontally. The remaining services, given that AWS manages them, would scale automatically, such as S3, or by increasing their instance sizes, such as RDS.

When considering the possibility of scaling the system, the respondent also identified the possibility of decomposing the application monolith to multiple services. He also described that the deployment could be customized so that the instance resources aligned with the service's requirements.

10.6.2 Infrastructure Management

There was no infrastructure automation. The AWS user interface was used to allocate and manage infrastructural resources. Given the static nature of the allocated infrastructure, INFRASTRUCTURE AS CODE was not adopted. Still, the importance of automation was recognized but postponed to when infrastructure changes became more frequent.

At the time, a single EC2 instance handled all traffic, but the architecture was already designed to scale horizontally, meaning that new instances could be added behind the allocated load balancer at any time to scale the system.

10.6.3 Orchestration and Supervision

The team adopted Deployer[9] for deploying the software, facilitating deployments over SSH to their allocated Virtual Machine (VM). Deployer executed in the application server, and when required, it was instructed to update the application on the server. A SSH command triggers its execution, which operates by pulling the latest source version from the master branch of a git repository, running the necessary migrations, and attempting to start the new version. The process would be rolled back automatically on failure. Deployer provided an improvement from manual deployments, which had to be done outside office hours and required several minutes of downtime. The responded stated that, at a larger scale, Deployer would still be used, but the deployment strategy would have to be incremental for the users, similar to a blue-green deployment [BP19].

CONTAINERIZATION was adopted for development to set up test environments with all their dependencies quickly. The respondent has not yet felt the need to adopt CONTAINERIZATION in the production environment. When faced with the possibility of having to scale their system, the respondent considered Docker as a better alternative than to manually configure the server to host the application.

There was no supervision in place. In the rare events when the team observed system failures, a team member would access the server to inspect and recover it manually.

[9] Deployer is a deployment tool of PHP. Learn more at https://deployer.org/

The respondent has never researched about ORCHESTRATION MANAGER and considers premature any effort in that direction. Cron was used in the allocated server for executing jobs periodically. The lack of an ORCHESTRATION MANAGER implementation discarded the adoption of JOB SCHEDULER.

10.6.4 Discovery and Communication

Given the monolithic nature of the application, there was no need for implementing the SERVICE DISCOVERY. RabbitMQ was being evaluated for adoption to implement an asynchronous MESSAGING SYSTEM to manage a queue of pending reports or sending emails when there was less CPU demand. In the future, this responsibility could be decoupled into a dedicated service and server.

10.6.5 Monitoring

Infraspeak used CloudWatch[10] and New Relic[11] for server monitoring. CloudWatch triggered alarms for the team using multiple channels. CloudWatch was used mainly for AWS service metrics, such as CPU usage in an EC2 instance.

New Relic was used for application-related metrics, measuring error response rates or response latency, and to monitor the public endpoints of the application. It was not possible to understand the client for which errors were happening in an initial development stage. The team has since been motivated to apply PREEMPTIVE LOGGING.

The two services together implement the LOG AGGREGATION and EXTERNAL MONITOR patterns, which the respondent considered essential for understanding the application's state through an unbiased observation quickly. As such, the respondent described that the typical process to identify failures is to first examine the status reported by CloudWatch for infrastructure and New Relic for application details.

10.6.6 Summary

Table 10.3 (p. 172) summarizes pattern adoption by Infraspeak. We can see that the team had mostly invested in their log management strategy, implementing LOG AGGREGATION ant PREEMPTIVE LOGGING patterns, facilitating their debugging process.

The team was considering adopting the MESSAGING SYSTEM to handle asynchronous tasks such as sending emails or generating reports during periods of less CPU usage.

[10] CloudWatch is a monitoring and management tool from AWS. Learn more at https://aws.amazon.com/cloudwatch/.

[11] New Relic is a commercial monitoring platform. Learn more at https://newrelic.com/.

Company name	Infraspeak
Company size	10 – 100
Active monthly users	100 – 1k
Product operation strategy	Single system, limited users
Pattern name	**Adoption**
INFRASTRUCTURE AS CODE	◐
AUTOMATED SCALABILITY	
CONTAINERIZATION	○
ORCHESTRATION MANAGER	
AUTOMATED RECOVERY	
JOB SCHEDULER	○
FAILURE INJECTION	
PREEMPTIVE LOGGING	●
LOG AGGREGATION	●
EXTERNAL MONITOR	●
MESSAGING SYSTEM	◐
SERVICE DISCOVERY	
Adopted patterns count	3

Table 10.3: Overview of the pattern language adoption by Infraspeak. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented.

Until the interview, Infraspeak had been able to operate its application using a single VM, manually allocated through the AWS web interface. Nevertheless, motivated by a recent investment round, the architecture had been redesigned to scale horizontally. The system was ready to be scaled into additional VM instances, which would live behind the already allocated load balancer. Such change can trigger the exploration of INFRASTRUCTURE AS CODE. For the future, the team is considering experimenting with microservices for optimized hardware allocation, which might lead to the implementation of CONTAINERIZATION.

10.7 Case Study: SwordHealth

SwordHealth provides a physical therapy application for patients to perform physical exercises right from their home. It uses a tablet to instruct the patient on how to perform the exercises and a multitude of sensors to monitor the exercise execution, ensuring that the patient makes the proper movements. Remote physical therapy is relevant for people

who live in remote locations with limited access to health care. The company is focused on the American market, given the limited access to therapists in remote locations and the high costs of health treatments.

The patient and his exercises can be followed remotely by a health professional, ensuring that the patient is working correctly, making the treatment nearly as efficient as possible if done in the presence of a professional. The application claims to be more efficient at measuring the patient's progress than a physical therapist. This is because the multiple sensors are very accurate and can continuously monitor the user's progress, detecting any minimal change in the patient's performance.

SwordHealth has published research that demonstrates that patients using their product daily show faster recovery than those relying on visiting physical therapists [Cor+18].

The interview with SwordHealth occurred on October 31, 2018, with Tiago Seabra, VP of engineering.

10.7.1 Product Overview

The product started as a monolith. At the time of the interview, it was undergoing the process of decoupling into microservices.

The decoupling was well thought and motivated. The first motivation was to take the opportunity to refactor the initial prototyping code that no longer met the team's quality standards. Development isolation was also sought after, to prevent failures from one component from interfering with others. It optimized the hardware required to scale the application, scaling only instances for the services which see their usage change during the day, without changing the ones that have a more stable demand. Dedicated storages were also identified as more straightforward to manage and migrate when needed, as opposed to using a single database for the whole product.

SwordHealth runs multiple instances of their product due to the medical nature of its data, which must comply with each country's privacy legislation. To increase the number of regions where they can deploy, they work with both Microsoft Azure and Google Cloud.

The largest installation of SwordHealth is yet to scale above the thousands of users, but they believe their user base will increase next year, motivated by the penetration in the North American market.

10.7.2 Infrastructure Management

Deployment pipelines were built using Jenkins pipelines. These created a software package from the source code implementing CONTAINERIZATION and then used INFRASTRUCTURE AS CODE to set up the required environment and deploy the containerized services on top of it. When the team checked source code into their git's master branch, which could only be performed by senior developers, the associated Jenkins pipeline automated the containerization of the services and updated the proper environments. Helm[12] was used to implement INFRASTRUCTURE AS CODE, being responsible for deploying software packages in Kubernetes instances. Helm improved on the use of the default Kubernetes client by facilitating the description of container dependencies or providing strategies for applying migrations, reverting the deployment if the migration failed.

SwordHealth instantiated independent databases and virtual servers for each instance of their application. Still, some supporting services were centralized and shared across all deployment instances. That was the case for Google's PubSub for MESSAGING SYSTEM, Google Cloud Storage for file storage, and Big Query for sensor data without Personal Identifiable Data (PII).

SwordHealth had a QA team that deployed testing environments to test the changes in a branch manually. The QA team triggered the deployment of the branch to test directly from Jenkins and had no interaction with the deployment process.

Due to insufficient code review practices and QA, migrations had failed in previous deployments to production environments. These have required manual intervention for recovery. That was the only deployment issue identified by the respondent. We can argue that this is a development issue rather than a deployment one.

10.7.3 Orchestration and Supervision

SwordHealth used managed Kubernetes for implementing ORCHESTRATION MANAGER, both in Azure and Google Cloud. Managed Kubernetes provides Kubernetes operated by the cloud provides, which also ensures AUTOMATED SCALABILITY without intervention from the team. Kubernetes also implemented AUTOMATED RECOVERY and JOB SCHEDULER. The team actively used both, configured through Helm.

The team experimented with the concept of FAILURE INJECTION by manually killing containers inside a cluster and verifying that they recovered automatically. Still, the process was never automated, nor was there any plan to do so.

[12] Helm is a Kubernetes package manager. Find more at https://github.com/helm/helm.

Services were configured to scale automatically when their CPU usage was above a given threshold. As described above, the infrastructure scaled automatically as well to accommodate these new services. Despite the support for AUTOMATED SCALABILITY, dynamic scaling was rarely observed given the reduced volume of users, easily handled by the minimum hardware allocation.

10.7.4 Discovery and Communication

Asynchronous processing and requests between services were deferred using a MESSAGING SYSTEM. All systems used a centralized Google PubSub service as a message queue. Asynchronous tasks ranged from sending welcome emails to new users, to generating user performance reports.

Within an application's instance, services needed to also communicate with each other synchronously. Kubernetes implemented SERVICE DISCOVERY with its internal Domain Name System (DNS) server, providing a unique domain name for each service. ORCHESTRATION MANAGER forwarded the traffic to the available instances using the configured routing rules.

10.7.5 Monitoring

StackDriver was chosen to implement both EXTERNAL MONITOR and LOG AGGREGATION. The team captured server metrics such as CPU, disk, or Random Access Memory (RAM) usage and configured alerts over those to detect unexpected parameters. StackDriver also queried the client-facing APIs in each geographic instance, generating notifications whenever a service was misbehaving. StackDriver tagged logs and metrics by source so that they can be mixed and matched for specific events that happened in specific services in a given date range and deployment instance. The respondent considered StackDriver limited regarding querying capabilities and considered migrating to an alternative in the future, such as Prometheus[13].

During development, each developer had the responsibility to decide the log verbosity level to adopt towards implementing PREEMPTIVE LOGGING. This decision was often promoted to a team-wide discussion.

Company name	SwordHealth
Company size	10 – 100
Active monthly users	100 – 1k
Product operation strategy	Single system, limited users
Pattern name	**Adoption**
INFRASTRUCTURE AS CODE	●
AUTOMATED SCALABILITY	●
CONTAINERIZATION	●
ORCHESTRATION MANAGER	●
AUTOMATED RECOVERY	●
JOB SCHEDULER	●
FAILURE INJECTION	
PREEMPTIVE LOGGING	●
LOG AGGREGATION	●
EXTERNAL MONITOR	●
MESSAGING SYSTEM	●
SERVICE DISCOVERY	●
Adopted patterns count	11

Table 10.4: Overview of the pattern language adoption at SwordHealth. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented.

10.7.6 Summary

Table 10.4 (p. 176) summarizes the pattern adoption by SwordHealth at the time of this interview. We have observed a great cloud maturity from SwordHealth, with all but one patterns from our language implemented. The need to maintain multiple instances geographically distributed, due to the privacy requirements from the health data the company captures, motivated an increased company maturity.

FAILURE INJECTION was the only pattern not implemented from the pattern catalog. The respondent told us that they would need an additional budget for investing in redundancy to be comfortable exploring the implementation of this missing pattern while mitigating the risk of impacting service continuity.

The respondent was pretty confident that their architecture would scale seamlessly given the need, without any change or intervention from the team, other than adjusting the size of their database instances.

[13] Prometheus is a centralized monitoring solution. Read more at https://prometheus.io/.

10.8 Case Study: Velocidi

Velocidi[14] helps direct to consumer brands to optimize their digital marketing budgets with their Customer Data Platform (CDP). The typical client would interact with hundreds of thousands of users monthly.

The interview with Velocidi took place on November 2nd, 2018, with lead engineer João Azevedo.

10.8.1 Product overview

Velocidi provided three core functionalities:

Data collection. brought client data into the system by using any real-time data from the interaction between the users and the brand. Data collection gathered data from web browsing, mobile application, offline shopping, or any other viable data point which could be used to enrich the user's profile. Data collection also enabled offline data imports, such as CRM data exports.

User segmentation. Facilitated the creation of user audiences. Audiences were lists of anonymous user IDs that should see the same marketing campaigns. Audiences were created using rules over any of the user data collected. An example of an audience could be one that targets male users that live in London are in an age band of 20 to 65 and browsed car websites.

Data activation. The product itself did not enable the client to buy or run marketing campaigns. For that purpose, the product sends user audiences to another marketing platform responsible for running advertising campaigns. Audiences sent to other marketing platforms as lists of anonymized user IDs such as email hashes or mobile phone advertising IDs.

A vital characteristic of the product is that it is by design a first-party platform. User audiences created with deterministic first-party data enabled the optimization of marketing campaigns, which has been proven to increase marketing Return On Investment (ROI) [Sim16]. As such, the company did not host the CDP for its clients or reuse data across clients, but instead, it provides the software to its clients through a managed license, with a dedicated deployment per client in his cloud account. Such guarantees that the

[14] Disclaimer: At the time of the interview, the author was employed by Velocidi. This research both influenced and was influenced by the implementation of Velocidi's product. Learn more about the Velocidi CDP at https://www.velocidi.com/.

ownership of data remains on the client-side, disregarding any need to have the data leave his premises, at the added cost of operating one CDP instance per client.

With a team of seven engineers and clients using the software to manage data for over 80 million user IDs in the most significant client, adopting best practices for cloud development was essential.

Being a startup with limited resources, the only tangible way to achieve such complex operation was through optimal architectural decisions and elaborated automation, allowing a small team of fewer than ten engineers to sleep well at night.

10.8.2 Infrastructure Management

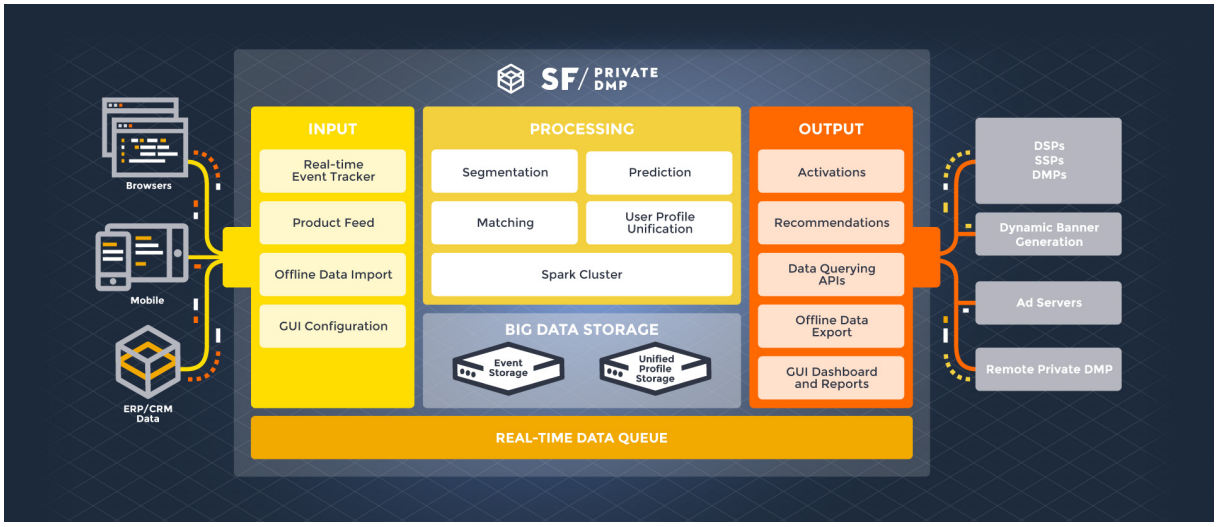


Figure 10.3: A graphical representation of the services that composes Velocidi CDP. On the left are the input services. In the middle, the data processing and storage services, and the outputs are on the right. This figure refers to SF Private DMP, the original product name before Shiftforward was acquired by the North American company Velocidi in late 2018.

The CDP was composed of a multitude of services that collaborate to provide the application as a whole. Figure 10.3 (p. 178) depicts its main services. Services on the left side are the services responsible for receiving data into the system. In the middle are services that provided persistent storage. On the right are the output services that make information available to third parties, generate reports or product recommendations, and much more.

Not all CDP instances required all these services, with each instance curated for the client's needs. The team deployed the CDP exclusively using AWS for the cloud provider. It did so by using their proprietary deployment tool, *Collie*, which was their implementation of INFRASTRUCTURE AS CODE. *Collie* relied on HashiCorp's Terraform for

most of its automation. A configuration file per client defined the required infrastructure and application pieces for him, which *Collie* deployed or updated.

While continuous integration was adopted to build all components automatically, deployment was still manually triggered using *Collie*. The respondent considered this a candidate for improvement in the future, mandatory for increasing the number of clients managed. Once deployed, the CDP relied on AWS's CloudWatch and Auto Scaling Groups to dynamically scale the infrastructure up or down according to resource usage and threshold policies.

10.8.3 Orchestration and Supervision

Docker provided CONTAINERIZATION, enabling isolated environments to be created once and deployed for any client. During development, the latest version of each service was packaged and uploaded to a private container repository, to which client cloud accounts could pull the images. Containers facilitated orchestrating services programmatically in different environments and clients with little to no added configuration. Environment variables provided the specific configurations that changed per client to configure the container. The team also creates client-specific configurations that are made available to the container in run-time.

Velocidi adopted an ORCHESTRATION MANAGER around 2012 by using Mesos and Marathon, moving away from Java ARchive (JAR) file deployments. Recently before the interview, the team migrated to Kubernetes. This change was motivated by the broader community and development speed of Kubernetes over Marathon. All new clients were deployed with Kubernetes, but the team continued to support earlier systems using Marathon, given the complexity of the migration.

Velocidi instantiated different types of machines and allocated services onto them based on the machines' specifications and the services' requirements. As an example, services that performed CPU intensive jobs were allocated to machines with better or more CPUs.

The ORCHESTRATION MANAGER provided AUTOMATED RECOVERY strategies that internally monitored and restarted services if needed. Restarting failing containers back to a clean state was often able to get the system functional again, even if only temporarily. As an example, the ORCHESTRATION MANAGER issued HTTP requests to the services, observing their response. If the expected result was not received, the ORCHESTRATION MANAGER restarted it.

AUTOMATED SCALABILITY was implemented using Amazon Web Services

Auto-Scaling groups, which adjusted the number of allocated virtual machine instances for the ORCHESTRATION MANAGER, taking into consideration CPU and RAM resource usage. New instances joined the cluster automatically, becoming available for allocating new services or service instances.

Chronos was used as a JOB SCHEDULER for Mesos for setting up scheduled tasks, typically by providing a container image and its execution constraints and environmental variables, like any other container. This way, Mesos could allocate the job in the cluster, and once the job completed, the container stopped and the resources made again available in the ORCHESTRATION MANAGER.

10.8.4 Discovery and Communication

Being a complex cloud application from the start, the CDP was initially designed to use microservices. Doing so introduced the need for services to communicate with each other. RabbitMQ was the adopted implemented MESSAGING SYSTEM to facilitate decoupling services and their communication. Routing keys were used to route traffic. When services connected to the MESSAGING SYSTEM, they described which keys they were interested in consuming. The use of queues to distribute work facilitated scaling specific microservices.

Some services had only to communicate with each other, so direct communication was preferred to using MESSAGING SYSTEM to increase performance and eliminate unnecessary latency and resource usage with the RabbitMQ indirection. SERVICE DISCOVERY provided a strategy for services to find an establish direct communication, using a reverse proxy with a known local service port used to forward traffic from one of the available instances of the destination service. Marathon-lb was the adopted implementation, which natively integrated with Mesos for service discovery and automatic port forwarding configuration.

An example of the two communication strategies described can be observed in the communication between the following services:

Event tracker. a public-facing service, responsible for handling browser requests that bring user events into the system;

Event Augmenter. receives user events and enriches them with additional information, such as geolocation data based on the user's IP address;

Event storage. stores and provides access to other services to all user events;

User profile storage. stores and provides access to other services to user profiles, namely, the user's identifiers and attributes.

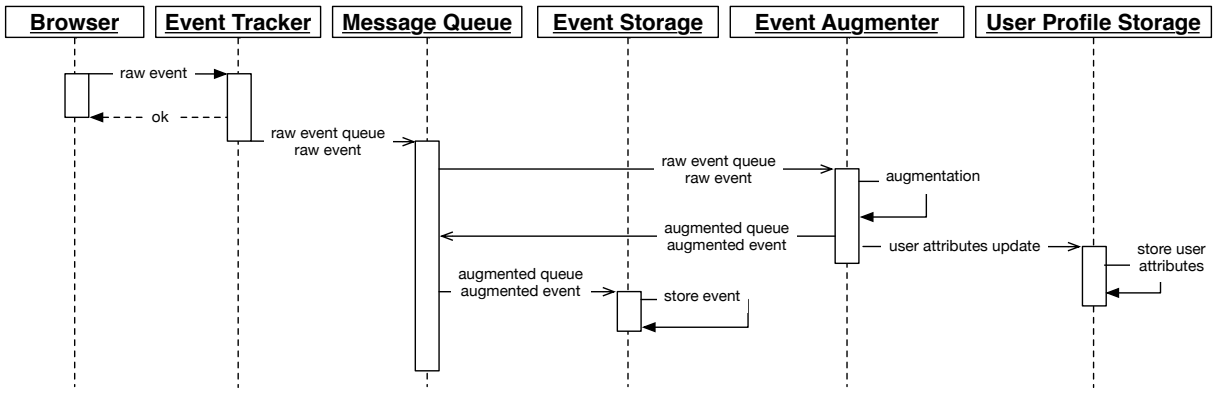


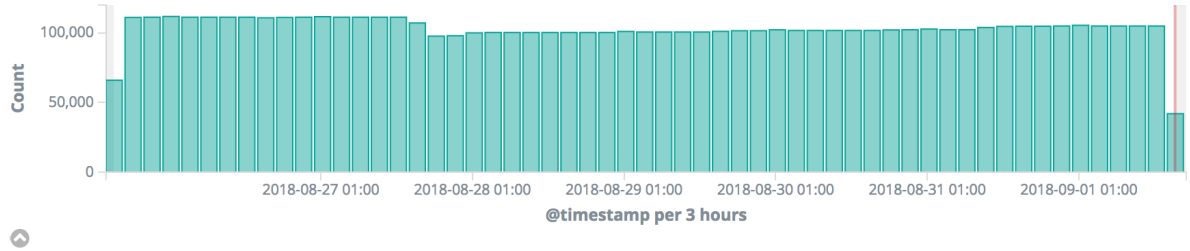
Figure 10.4: Sequence diagram representing the communication between services in the CDP, using a MESSAGING SYSTEM or a SERVICE DISCOVERY.

Figure 10.4 (p. 181) demonstrates how the services described above cooperate. A sequence diagram demonstrates the flow of data between them. The *Event Tracker* service had an HTTP server that listened for browser requests, which, in this case, consisted of browser events from the client. These events were sent to the MESSAGING SYSTEM. An *Event Augmenter* service (not represented in Figure 10.3 (p. 178)) was listening in the *raw events* queue from the MESSAGING SYSTEM, consuming all events, enriching them with a new set of attributes according to its configuration. The *Event Storage*, represented in the middle, was listening for augmented events from that queue and persisting them permanently. These were then sent to the *User Profile Storage*, using its HTTP API, which was responsible for the permanent storage of user data. Given that the *Event Augmenter* would only communicate with the *User Profile Storage*, their communication was direct, facilitated by the adoption of SERVICE DISCOVERY with a different service port for each service deployed.

10.8.5 Monitoring

Services forwarded their execution logs to an ElasticSearch database using LogStash[15], implementing the LOG AGGREGATION pattern. Kibana, a visualization tool and part of the Elastic Stack, provided a user interface for navigating the logs and the possibility of creating data visualization dashboards. Such dashboards could be leveraged to identify abnormal event patterns in the application. An example of visualization over the

[15] ElasticSearch is an open-source full-text search database. LogStash is a server-side log processor that transforms the logs and forwards them to persistent storage, in Velocidi's case, ElasticSearch. Both are part of the Elastic Stack, a set of open-source tools to enable the capabilities of ElasticSearch. Read more about them at https://www.elastic.co/



Time	sourceName	avg	metricName
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	114,294,784	jvm.memory
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	1,073,741,824	jvm.memory
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	83,886,080	jvm.memory
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	89	host.cpu
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	30,596,573	jvm.memory
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	10,354,688	jvm.memory
September 1st 2018, 16:11:00.008	adstax-customer-lifetime-value	32,899,072	jvm.memory

Figure 10.5: Screenshot of a Kibana's metrics visualization for seven days. The table allowed expanding each event individually to its details. The visualization could be filtered by date, source or event details.

aggregated data from a system is depicted in Figure 10.5 (p. 182). ElastAlert[16] was used to identify issues in the captured metrics and raise alarms to the development team, usually sent via email. The whole monitoring stack was deployed for each client, ensuring proper data separation.

StatusCake[17], a service that monitors uptime and performance over HyperText Transfer Protocol (HTTP) was used for implementing the EXTERNAL MONITOR pattern. During the infrastructure orchestration and service deployment, StatusCake tests were set up to verify that public endpoints were responsive continuously.

The respondent highlighted the importance of implementing EXTERNAL MONITOR to identify network misconfigurations, such as wrong firewall rules, that could prevent access to the system from outside its network.

FAILURE INJECTION had not been implemented given the required level of investment in both experimentation and infrastructure redundancy that could cope with the random injection of failures, but it has been considered for the future.

[16] ElastAlert is another application from the Elastic Stack, facilitating the creation of alarms out of the captured data.

[17] StatusCake is a commercial monitoring tool. Learn more at https://statuscake.com.

10.8.6 Summary

Company name	Velocidi
Company size	10 – 100
Active monthly users	> 1M
Product operation strategy	One system per customer
Pattern name	**Adoption**
INFRASTRUCTURE AS CODE	●
AUTOMATED SCALABILITY	●
CONTAINERIZATION	●
ORCHESTRATION MANAGER	●
AUTOMATED RECOVERY	●
JOB SCHEDULER	●
FAILURE INJECTION	◐
PREEMPTIVE LOGGING	●
LOG AGGREGATION	●
EXTERNAL MONITOR	●
MESSAGING SYSTEM	●
SERVICE DISCOVERY	●
Adopted patterns count	11

Table 10.5: Overview of the pattern language adoption by Velocidi. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented.

Table 10.5 (p. 183) summarizes the patterns that Velocidi has applied during the development of their CDP. We could observe a level of maturity similar to the one observed with SwordHealth with Velocidi, possibly also motivated by the need to deploy one instance of their product per client, with the increased challenge of doing it in the client's cloud provider account. Velocidi implemented eleven out of the twelve in the pattern language. Again, we found that FAILURE INJECTION was not implemented, given the prohibitive infrastructure costs required for redundancy to implement the pattern.

To scale the number of clients, the respondent highlighted the need to enable continuous deployment of their software for all client accounts, as manual deployments, despite supported by INFRASTRUCTURE AS CODE, would become unsustainable.

	Company name					
	LabOrders	**HUUB**	**Infraspeak**	**SwordHealth**	**Velocidi**	**Count**
Company size	1 - 10	10 - 100	10 - 100	10 - 100	10 - 100	
Active monthly users	1 000 - 10 000	1 - 100	100 - 1 000	100 - 1000	> 1 000 000	
Product operation strategy	Single system, limited users	Single system, limited users	Single system, limited users	Single system, limited users	One system per customer	
INFRASTRUCTURE AS CODE	○	●	◐	●	●	4
AUTOMATED SCALABILITY		◐		●	●	2
CONTAINERIZATION	◐		○	●	●	3
ORCHESTRATION MANAGER	◐			●	●	2
AUTOMATED RECOVERY				●	●	2
JOB SCHEDULER		○	○	●	●	4
FAILURE INJECTION					◐	0
PREEMPTIVE LOGGING	●	◐	●	●	●	4
LOG AGGREGATION	○	○	●	●	●	5
EXTERNAL MONITOR	○		●	●	●	4
MESSAGING SYSTEM		●	◐	●	●	3
SERVICE DISCOVERY				●	●	2
Adopted patterns count	1	2	3	11	11	

Table 10.6: Overview of the pattern language adoption by the interviewed companies. ● is used when the company has implemented the pattern; ○ is used when the pattern or a variation from it is partially implemented, or implemented with limitations; ◐ when the company is considering implementing the pattern and an empty space for when no effort has been started to implement the pattern. The last row provides the count of the patterns each company has fully implemented, while the last column provide a count over how many companies fully or partially have implemented a pattern.

10.9 Discussion

This chapter described a case study over five companies whose main activity is developing a cloud product, evaluating their cloud architecture and practices, and relating them with the pattern language from Chapter 6 (p. 69). We aim to understand how the interviewed companies apply the pattern language, what motivates them to do so, and, if possible, capturing additional details for improving the language. Our findings are summarized in Table 10.6 (p. 184).

LabOrders. The smallest of the five interviewed companies regarding the number of users and team size. Their product served a small number of users and did not have to scale dynamically. At the time, they were operating out of a single server, which rarely needed to change and had basic operational requirements. They adopted PREEMPTIVE LOGGING to help capture runtime issues with sufficient detail for the team to address them. Still, this information was not readily available to the team, so they considered LOG AGGREGATION essential at a larger scale. To attain such a scale, they were considering moving towards microservices and adopting INFRASTRUCTURE AS CODE to enable scaling and facilitate operations, respectively. To overcome their timely and error-prone manual update process, they considered implementing CONTAINERIZATION to package their application and ORCHESTRATION MANAGER to run them at scale when needed.

HUUB. Huub was similar in operations strategy and user count as LabOrders. The team was observing exponential growth, accelerated by the external investment they had just received. An expanding engineering team enabled refactoring their initial code, and, with it, they were starting to align with our pattern language. At the time, they had already implemented INFRASTRUCTURE AS CODE for managing their infrastructure, and MESSAGING SYSTEM, which enabled deferring computational work outside business hours. Implementing AUTOMATED SCALABILITY was their current priority, given the expected continued growth, followed by PREEMPTIVE LOGGING for facilitated debugging of issues observed in production.

Infraspeak. Already at a later business stage than the previous two, at the time, Infraspeak was expanding its market out of Europe to Africa and South America at the time. While their software was still a monolith, a recent financial investment enabled an increase in their team size, which bootstrapped the plans for decoupling part of the logic into independent microservices. During this process, they have implemented PREEMPTIVE LOGGING along with LOG AGGREGATION for

facilitated observation of issues in production. They also monitored the system using EXTERNAL MONITOR. The introduction of microservice was motivating the adoption of MESSAGING SYSTEM to distribute asynchronous work, such as sending email notifications. They were considering adopting AUTOMATED SCALABILITY and INFRASTRUCTURE AS CODE for improved operations.

SwordHealth and Velocidi. These two companies presented a similar level of architecture design maturity. Both companies operated multiple instances of their product, deployed to different cloud regions or providers, mostly due to privacy restrictions. Manage one product instance per client difficulted operations, demanding automation across infrastructure management, discovery and communication, and monitoring, leading both companies to adopt all but one pattern from our language. In both cases, FAILURE INJECTION was not implemented, justified by the prohibitive costs required to provide the redundancy required to cope with random deletions of resources in the infrastructure to evaluate its automated recovery.

The maturity from SwordHealth and Velocidi provided valuable input to improve some of our patterns. SERVICE DISCOVERY changed the most, by extending the initial version with adding DNS as a discovery strategy, along with the already described local reverse proxy strategy.

10.10 Conclusions

From the interviews over the five companies, we reached the following conclusions:

Team maturity influences pattern adoption. In Section 10.1 (p. 150), we hypothesize that there might be **a correlation between the maturity of a company and the number of patterns they adopt**. This case study does show that **a company's maturity influences pattern adoption**, with companies with more experience and more sophisticated products gravitating towards adopting an increased number of patterns.

No premature optimization. Infraspeak and LabOrders were not preemptively optimizing their cloud approach. They considered that their current architecture was able to address the volumes at which they were operating and felt no need to optimize their cloud environment preemptively, despite recognizing the need

to change their design and operations strategies to cope with a more extensive operation scale.

Some patterns are likely more relevant than others. This small sample enabled us to hypothesize on the relevance of each pattern for companies at different phases, which could hint at an ideal implementation order. LOG AGGREGATION is the only pattern addressed by all companies, which, in a cloud environment, is trivially explained, as it is troublesome to access each allocated resource to evaluate its logs. PREEMPTIVE LOGGING quickly follows, implemented by four companies, and under evaluation by HUUB. The pattern is trivial to implement and can go a long way into ensuring relevant information is available when an issue is observed in production, capacitating the development team with the required knowledge to address the issue. INFRASTRUCTURE AS CODE enables teams to prevent manual operation errors and avoid time-consuming manual operations. EXTERNAL MONITOR provides a holistic observation of the system, ensuring that the external user has the intended experience.

failure injection is a likely outlier in the pattern language. FAILURE INJECTION was not implemented by any company, showing that its relevance is likely not on par with the other patterns from the language. The main reason for not addressing this pattern was the lack of financial resources demanded to provide the required redundancy to implement it.

10.11 Threats to Validity

The use of empirical methods for validation is not without shortcomings and can be subject to multiple biases. This section identifies the most important ones and how we address them in this work.

10.11.1 Internal Validity

Personal contacts. All but one respondents were personal contacts and shared an engineering background with the interviewer. Such a relationship can lead to a relaxed interview environment, which could lessen the respondent's focus and result in overlooking relevant details. Also, the shared educational background can lead to similar strategies while designing software. We addressed this problem by designing the interview as a SSI to ensure a steady pace and maintain the interviewee focused

on the interview. We further address this threat with the complementing research strategy described in Chapter 11 (p. 191), which surveys a broader audience, with little to no personal contact. An ideal scenario would see the case study expanded to additional interviews with a more randomized set of interviewees.

Interviewer and respondent bias. Interviewer experience influences the application of SSI, along with the tacit domain knowledge from both the interviewer and respondent. If either part uses terminology that the other is not familiar with, there is a degradation of the communication, reducing the overall efficiency of the interview [LA94]. All interviews were performed by the author, who had a deep understanding of the domain, minimizing the risk of losing information from the interview.

Interview Protocol. The use of SSI might lead the respondent to focus on the questions asked, disregarding relevant information not directly asked, but might be relevant for the research. In this regard, we have asked the respondents at the beginning of the interview to diverge from the questions asked, encouraging them to introduce relevant related details about their product, architecture, or practices that might be anyhow related to the overall goal.

Information missed during the interview. The use of SSI with a single interviewer might render the interviewer unable to cope with the volume of information being shared. The audio from the interviews was recorded with the interviewees' permission to prevent losing relevant information, allowing a complete revision of the whole interview while analyzing it.

10.11.2 External Validity

Sample size and quality. Semi-structured interviews are very time-consuming. During this research, we have interviewed five respondents, all operating from the same city and with a similar engineering background, which provides limited statistical significance and struggles to demonstrate the accuracy and completeness of the pattern language per se within the whole industry. Statistically stronger results require interviewing a larger sample, to which the necessary resources were not available for this dissertation's scope. We try to overcome the limited sample observed with the research described in Chapter 11 (p. 191), which asked similar questions to the ones in this case study in a survey responded by over a hundred professionals.

Geographic distribution. All interviewed companies are from the greater Porto region, in Portugal. Local company and technological culture might influence all these professionals, rendering a biased observation of the overall industry reality. We address this limitation once more with the research described in Chapter 11 (p. 191), which forwards the survey to professionals from geographic regions scattered around the world.

10.12 Summary

We have described a case study of five companies developing their software for the cloud, observing that all companies implement at least some patterns from the pattern language. We could correlate the companies' cloud maturity with the number of patterns they adopt. We have observed that the two companies that needed to operate multiple instances of their product have implemented all patterns from the language but one, FAILURE INJECTION. We hypothesize this would be due to the unmanageable cost of redundancy, for most companies, required to implement the pattern.

Despite the limited population used for this case study, the level of detail to which we were able to discuss these use cases provided valuable insight into the reality of these companies, enabling us to infer what struggles other companies might face as well. This knowledge enabled the improvement of the pattern language with new forces and implementation details, mostly for SERVICE DISCOVERY. We also verify that the industry is aware of these patterns and that they are relevant for designing cloud software at any stage, with new patterns implemented as the product matures.

Alternative validation strategies are still required, given the limited sample in this case study. A more extensive survey would help us understand how widespread and essential these patterns are. We address this in Chapter 11 (p. 191).

During this chapter, we address the following RQs:

RQ3. What driving forces influence how strategies are implemented?

While interviewing Velocidi, we have observed they their implementation of SERVICE DISCOVERY was different from ours, which enabled us to improve the pattern description with new forces and implementation details.

RQ4. Are companies that develop software for the cloud aware of these problems and adopt the identified solution?

We observe that companies adopt the patterns from the pattern language to design their cloud software. In the specific case of SwordHealth and Velocidi, they

implement all but one pattern, verifying the relevance of these patterns for the industry.

Chapter 11

Pattern Language Adoption Survey

During this dissertation, we identify recurring design practices that facilitate the development of cloud software. These practices were captured using patterns and described in Chapters 6 to 9 (pp. 69, 77, 117 and 135). The conclusions from the case study described in Chapter 10 (p. 149) further helped improve the pattern language, adding new forces and implementation details to the patterns described in the previous chapters from the experience of the interviewed companies. The previous chapter describes a case study with five companies approaching cloud development and how they relate to the identified patterns. From that limited population size, we observed that more mature companies tend to adopt an increased number of patterns in the language. Would we still find the same with a broader audience? Could there be a correlation between a company's characteristics and the number of patterns they implement? This chapter describes a survey of over 100 professionals, whom we inquire about their company characteristics and pattern adoption, and evaluate the existence of correlations between those.

11.1 Goals

The previous chapter presented insight regarding how companies build their software using a Semi-structured interview (SSI) in a case study with the cooperation of five companies. Due to time constraints, we were unable to conduct case studies with a broader audience. Nevertheless, we want to understand how a wider audience is aware and implements our pattern language.

We look forward to addressing RQ2 and RQ4 further, by asking respondents to identify which strategies they apply, and RQ5, by classifying each response regarding three key company characteristics which we attempt to then correlate with how often each pattern is implemented.

11.2 Methodology

We used an online questionnaire to reach cloud professionals and gather quantitative and qualitative data without requiring an interviewer. Online questionnaires facilitate the dissemination of surveys without geographical barriers or temporal limitations [Pun+03]. We applied the guidelines from the Web Survey Methodology [CMV15], which provide extensive details on how to create intuitive online questionnaires and minimize nonresponse.

Given that the respondents were volunteers, we took into account that their availability and willingness to spend significant time answering the questionnaire was limited. We considered the number of questions and the time required to answer them, to reach a balance that would allow acquiring as much knowledge as possible, while shortening the time to respond. All questions were mandatory for submitting the form.

11.2.1 On Using Questionnaires

Contrary to interviews, questionnaires facilitate gathering data without the need for an interviewer, requiring little time or money to reach the respondents [Owe02]. Web questionnaires specifically facilitated the dissemination of the survey without geographical and with less temporal limitations.

In this research, we followed guidelines from the Web Survey Methodology[1], which

[1] WebSM is a website maintained by the Faculty of Social Sciences from the University of Ljubljana, aggregating bibliography and resources for creating and running surveys using the Web. Learn more at http://www.websm.org/.

provides extensive details on how to create intuitive online questionnaires and minimize nonresponse.

Given that the respondents were volunteers, we took into account that their availability and willingness to spend significant time answering the questionnaire was limited. In that regard, we balanced the number of questions with the time required to answer them, to reach a balance that would allow extracting as much knowledge as possible, while keeping the time required to respond short. We did not accept partial responses, and all questions were mandatory.

11.2.2 Target Audience

We targeted with this survey professionals building software for the cloud. The questionnaire is trivial to reply to and includes only high-level questions, so we expect most people familiar with the cloud architecture of their company to respond to them.

11.2.3 Variable identification

As we want to assess how widespread the patterns from the language are adopted in the industry, we specifically asked if the respondent adopted each pattern and measured their positive and negative answers. The *pattern adoption* is the dependent variable in this study, and to characterize the respondents we used three independent variables: *product operation strategy*, *active monthly users*, and *company size*.

Product operation strategy pertains to how the software is deployed and operated. These strategies are ordered by level of complexity, as operating additional customers and independent systems will likely correlate with more complex and larger designs. They are:

- *Single system, single customer (SSSC).* A single deployment operated for a well-known number of users. *E.g.*, a corporate email server often has a well-known scale of operation and does not have to be scaled dynamically, which makes this strategy the simplest to operate.

- *Single system, multiple customers (SSMC).* The application runs in a single instance that has to adapt to a variable volume of customers, who can register themselves at any time. Examples of such applications are Netflix or Facebook. The system needs to scale dynamically to accommodate variations in traffic throughout the day.

- *One system per customer (OSPC).* A private deployment for each customer. The scale at which each instance operates can itself be dynamic. The number

of independent systems needs to be scaled, and each one has to keep scaling as well, typically through automation, *i.e.*, without scaling an operations team proportionally to the number of instances, which makes this the most complex strategy to operate.

It is common to measure how large a cloud application is by evaluating its number of **active monthly users** [AM17; Hub+13], which is the number of users that interacted with the application at least once during a month. More users will result in increased traffic, hence, demanding more complex operations supported by an appropriate infrastructure. We could not identify a standard to measure **active monthly users**, so we decided to use orders of magnitude with the intervals: (a) *less than 100*, (b) *100 – 1k*, (c) *1k – 10k*, *10k – 100k*, (d) *100k – 1M*, and (e) *more than 1 million*.

The third independent variable is **company size**. We analyzed how pattern adoption varies considering the number of employees in the company, according to the European Commission definition for company sizes [Eur15], which classifies companies as *micro*, *small*, *medium*, or *large*, according to their number of employees being 1 to 10, 11 to 50, 51 to 250, and over 250 employees, respectively.

11.2.4 Questionnaire

We organized the questionnaire into three categories. Company categorization (CC), with three questions, categorized the respondent's workplace. Pattern adoption (PA), with twelve questions, described which patterns the respondent had implemented in his workplace. Respondent classification (RC), with six questions, classified the respondent in terms of geography, company size, and role in the company.

If not otherwise specified, the questions had the following possible response options: (1) Yes, for the entire system; (2) Yes, partially; (3) Under assessment or development; (4) No, but we would like to; (5) No, it is not relevant; (6) I don't know I don't want to answer.

We describe the questions below, along with their rationale. The original questionnaire and its responses are available in Appendix B (p. 239).

Company Categorization

CC1. Product Operation Strategy Consider how you deploy your software, regarding the users that it is intended for.

Options: (1) OSPC – Private deployment for each customer, private to his users;

(2) SSMC – shared platform for any customer, e.g. Netflix, Facebook; (3) SSSC – only one deployment for a specific group of users.

Rationale: This question evaluates the complexity of the respondent's operations. We categorized product deployment strategy into three options:

Single System for Single Customer (SSSC). The team handles a single deployment and operates it for a well-known number of users. An example of such a strategy would be a corporate email server, where it is well known when new users are added to the system. Knowing the scale at which an application operates and not having to scale it dynamically makes this strategy the simplest to operate.

Single System Multiple Customers (SSMC). The application runs in a single instance that has to adapt to a variable volume of customers, who can register themselves at any time. Examples of such applications are Netflix or Facebook. The system needs to scale dynamically to accommodate variations in traffic throughout the day.

One System Per Customer (OSPC). A private deployment for each customer, to be used exclusively by his users. The scale at which each instance operates can itself be dynamic. Velocidi and Swordhealth adopt these strategies, viz Section 10.8 (p. 177) and Section 10.7 (p. 172). The team needs to scale the number of independent systems they manage, as well as keeping each one of those systems scaling as well, typically through automation, that is, without scaling an operations team proportionally to the number of instances, which makes this the most complex strategy to operate.

CC2. Active Monthly Users Consider the average number of users for your platform, across all system instances.

Options: (1) < 100; (2) 100 - 1 000; (3) 1 000 - 10 000; (4) 10 000 - 100 000; (5) 100 000 - 1 000 000; (6) > 1 000 000.

Rationale: The infrastructure sizing will typically be proportional to the number of active monthly users, with this number being a common indicator of scale with startup companies.

CC3. Adopted Cloud Providers Which of the following cloud providers are you using?

Options: (1) Amazon Web Services; (2) Google Cloud; (3) Microsoft Azure; (4) Private Infrastructure; (5) Hybrid / Multi-cloud; (6) Virtual Private Server.

Rationale: While indicators for cloud providers adoption are already available in Chapter 3 (p. 25), this question enabled us to verify how the respondents were distributed in terms of provider adoption.

Pattern adoption

P1. Have you adopted infrastructure as code? You automate your infrastructure and deployment operations programmatically. Example: Terraform, chef, ansible.

P2. Have you adopted automated scalability? Your system scales dynamically to adjust to elastic traffic. Example: Amazon Web Services (AWS) Elastic Compute Cloud (EC2) Auto Scaling.

P3. Have you adopted containerization? Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself. Containerization proposes the usage of containers to package the service and its dependencies and enable its isolated and programmatic deployment. Example: Docker.

P4. Have you adopted an orchestration manager? Deploying and updating software at scale is an error-prone, slow and costly process. Such can be facilitated by adopting an Orchestration Manager to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements. Example: Kubernetes, Mesos + Marathon.

P5. Have you adopted automated recovery? Services may fail during execution and need to be recovered in a timely and orderly fashion. Including health checks and recovery configurations in the instructions used for the Orchestration Manager to orchestrate containers, enables it to monitor and recover failing containers. Example: Kubernetes Pod Lifecycle.

P6. Have you adopted a job scheduler? Cloud applications require frequent short-running jobs to be scheduled, which must be orchestrated across a dynamic infrastructure without permanently allocating resources. A scheduler service running along with the Orchestration Manager can instruct it to allocate one time or periodic jobs, recovering their resources to the infrastructure when they complete. Example: Kubernetes Cronjobs.

P7. Have you adopted service discovery? Services might lack the network information required to communicate with other dynamically allocated services.

Communication can be achieved by abstracting service network details by relying on an external mechanism that facilitates communication and balances traffic between two services. Example: Kubernetes DNS, Marathon reverse proxy.

P8. Have you adopted a messaging system? As service instances increase, communication between services needs to be abstracted, enabling proper balancing between instances. This communication strategy is required to be fault-tolerant and scalable to maintain the application's resiliency. As a solution, a messaging system, colloquially known as message queue, can abstract service placement and orchestrate messages with multiple routing strategies between them. Example: RabbitMQ, Kafka.

P9. Have you adopted failure injection? Resilience mechanisms are triggered when software is failing. Since systems are designed to work correctly, the status quo prevents us to from continuously verifying the correctness of those mechanisms. We need additional strategies to minimize the probability of failure in production due to faulty resilience strategies. Failure injection software can generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms, verifying the application's resilience. Example: ChaosMonkey from Netflix.

P10. Have you adopted preemptive logging? The information required to debug issues in software is often lost during their first occurrence due to insufficient log verbosity. By adjusting logging verbosity preemptively in services and servers within acceptable resource limits (Central Processing Unit (CPU), storage, others), the team maximizes the probability of capturing relevant information for addressing future issues right from their first occurrence.

P11. Have you adopted log aggregation? Services orchestrated at scale produce disperse logs, resulting in a troublesome process to acquire and correlate those who come from multiple sources. This pattern suggests the Aggregation and indexing all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs. Example: Kibana, GrayLog.

P12. Have you adopted external monitor? Monitoring an application from inside the infrastructure that hosts it will result in an incomplete and biased version of the reality, for example, given the inability to observe issues such as lack of Internet connectivity or abnormal latency to the application. External Monitoring suggest

testing the application's public interfaces from an external source, providing an unbiased awareness of the application's status. Example: StatusCake, Pingdom.

Respondent Characterization

RC1. Your role in the product development

Options: (1) CTO; (2) CEO; (3) CIO; (4) Senior Engineer; (5) Engineer; (6) Architect; (7) Team leader; (8) Other.

Rationale: This response enabled the classification the respondent's involvement in the product development.

RC2. Number of collaborators in the company

Options: (1) 1 - 10; (2) 11 - 50; (3) 51 - 250; (4) > 250.

Rationale: The size of the company enabled us to infer the availability of human resources, while possibility a co-relation between company size and pattern adoption. The company sizes are adopted from the European Commission's company thresholds for micro, small, medium, and large companies [Eur15].

RC3-6. Company Name, Country, and Comments Rationale: This information further enabled classifying the respondent.

11.2.5 Design and Execution

The questionnaire was organized into two sections. The first part characterized the respondents and their product, asking about their active monthly users, company size, and operation strategy. Next, we asked if the respondent adopted each one of the patterns described in Chapter 6 (p. 69). The questionnaire was built with Google Forms. It was first piloted with fellow researchers and two cloud professionals, to optimize its clarity and ensure the relevance of the gathered data. Once improved, it was disseminated via email to software engineers, social networks, online forums, and local communities of cloud professionals. The recipients were asked to answer the questionnaire and forward it to their peers. The form was online from 2018/12/18 to 2019/1/29, during which time we got 102 responses. The collected data is publicly available online [SFC20].

11.3 Data Analysis

We start by characterizing the respondents and their companies (Section 11.3.1 (p. 199)). Next, we assess the relevance of each pattern by how often it is implemented by

professionals and by the intent to adopt it in the future (Sections 11.3.2 and 11.3.3 (pp. 199 and 202)), and infer relationships between them (Section 11.3.4 (p. 203)). We then analyze how pattern adoption varies according to the three independent variables that characterize the company and discuss the statistical significance of our findings using a Student's t-test and Analysis of Variance (ANOVA) (Sections 11.3.5 to 11.3.7 (pp. 205, 207 and 210)).

11.3.1 Respondents Characterization

Most respondents had a *technical role* in the company (82%), with a very balanced distribution regarding *company size*. Geographically, most of the answers we received were from Europe (70%), although a significant chunk of respondents refused to disclose their geographical location (perhaps due to privacy concerns, or because they regarded the company as international). We seem to have collected a reasonably balanced distribution of the number of active monthly users, although the order of magnitude *more than one million users* was (understandably) slightly under-represented. Figure 11.1 (p. 199) summarizes the respondents' characterization.

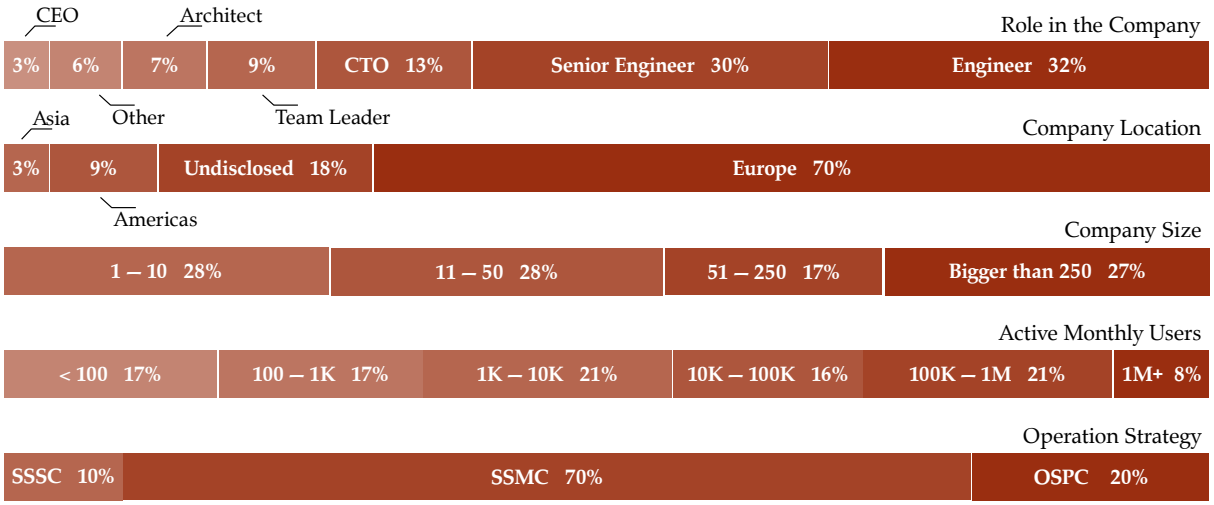


Figure 11.1: Demographic distribution of the survey.

11.3.2 Overall Pattern Adoption

Table 11.1 (p. 200) shows how frequently respondents have implemented each pattern. Figure 11.2 (p. 201) shows the adoption rate in terms of the number of patterns.

From this data, we make the following observations:

Pattern name	Adoption per pattern (%)
LOG AGGREGATION	72
MESSAGING SYSTEM	71
INFRASTRUCTURE AS CODE	69
CONTAINERIZATION	67
AUTOMATED SCALABILITY	65
JOB SCHEDULER	59
EXTERNAL MONITOR	57
ORCHESTRATION MANAGER	49
AUTOMATED RECOVERY	48
SERVICE DISCOVERY	47
PREEMPTIVE LOGGING	44
FAILURE INJECTION	17
Mean pattern adoption	55.88 ± 15.82

Table 11.1: Percentage of respondents adopting each pattern, sorted by most adopted. The bar on the right graphically represents the proportional adoption of each pattern.

- **The mean pattern adoption is 56%**, which translates to a reasonable coverage of the pattern language, asserting its relevance amongst practitioners;
- **97% of the respondents adopt at least one pattern**, with three respondents claiming to not implement any: (1) one uses SSMC and manages *10k – 100k* monthly users in a company with less than ten employees, (2) another operates a SSSC and manages $<$ *100* monthly users in a company with over 250 employees, and (3) the last one consists of a SSMC for $<$ *100* monthly users in a company with over 250 employees. Although we believe these to be *possible outliers*, **we do *not* discard them from our analysis**;
- **7 out of 12 patterns have an adoption rate of over 50%**, specifically LOG AGGREGATION, MESSAGING SYSTEM, INFRASTRUCTURE AS CODE, CONTAINERIZATION, AUTOMATED SCALABILITY, JOB SCHEDULER and EXTERNAL MONITOR;
- **log aggregation is the most adopted pattern.** Just like in any other software engineering domain, software is prone to failure. As the scale increases, so does the complexity needed to understand how systems behave. Debugging can range from being troublesome to nearly impossible without a platform that aggregates data from all components producing relevant execution metrics, justifying LOG AGGREGATION as the most adopted pattern from the language;

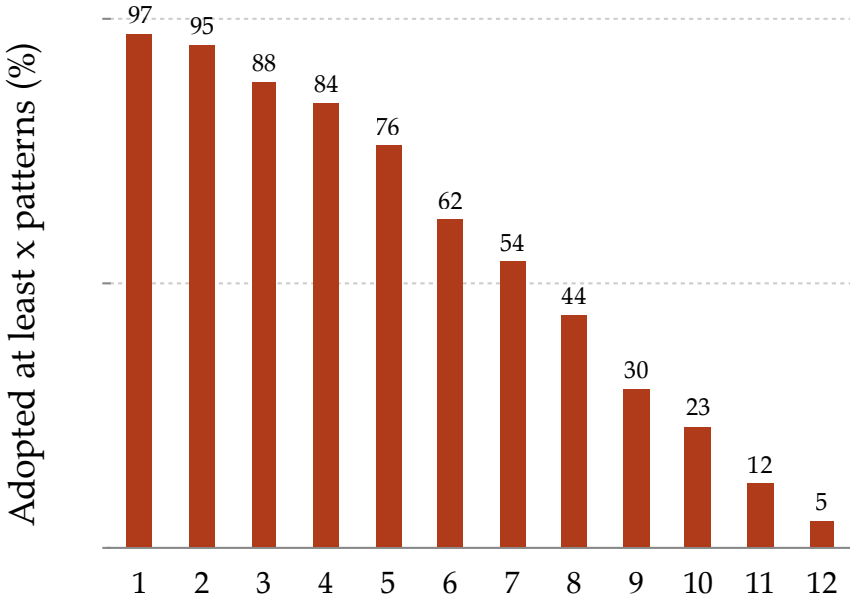


Figure 11.2: Adoption rate (in percentage) of the number of patterns. 97% adopted at least one pattern, 62% half of the patterns, and 5% all the patterns in the language.

- **messaging system is adopted by 71% of the respondents for asynchronous work distribution**, confirming its popularity to orchestrate messages between services, facilitating their cooperation via message passing and offloading complexity to a known *middleware*;
- **infrastructure as code closes the top three most adopted pattern list with 69%.** Manual interaction in operations increase their error-proneness, which justifies the increased adoption of this pattern to facilitate an infrastructure that nowadays tends to be large, heterogeneous, and elastic;
- **failure injection is the least adopted pattern**, with just 17% of the respondents using it, which deviates from the mean adoption by over two standard deviations (three if we calculate the mean without considering it). We conjecture that the lower adoption of FAILURE INJECTION could be due to its inherent complexity, the substantial investment required in hardware redundancy and supervision for its implementation, and the fact that the decision to deliberately increase the failure of a running system (even if in the long-run it would decrease it) is a hard-selling point.

11.3.3 Intent to Adopt

In a yes/no questionnaire, a negative response can be due to a multitude of reasons. We have added an additional question for those that answered *no* to each pattern adoption, attempting to distinguish between (a) those that eventually *want* or *intend* to use it in the future (*e.g.* the adoption is under study or under development), from (b) those that thought the pattern did not apply or was regarded as irrelevant to their system. Table 11.2 (p. 202) summarizes the data we collected, from which we make the following observations:

Pattern name	Intention (%)	Irrelevance (%)
LOG AGGREGATION	57.14	42.86
MESSAGING SYSTEM	44.83	55.17
INFRASTRUCTURE AS CODE	58.06	41.94
CONTAINERIZATION	66.67	33.33
AUTOMATED SCALABILITY	62.86	37.14
JOB SCHEDULER	51.22	48.78
EXTERNAL MONITOR	44.19	55.81
ORCHESTRATION MANAGER	71.15	28.85
AUTOMATED RECOVERY	71.70	28.30
SERVICE DISCOVERY	42.59	57.41
PREEMPTIVE LOGGING	56.14	43.86
FAILURE INJECTION	64.29	35.71

Table 11.2: Respondents that, while not adopting a pattern, *wanted* or *intended* to do it in the near future. The second column is the converse probability of the first, which translated in the questionnaire to the option of the respondent considering it *irrelevant* or *not wanting* to use it.

- **Though failure injection is the least (17%) adopted pattern, 64% intend to use it in the future,** making it the fourth most considered pattern for adoption;
- **On the other hand, service discovery not only ranks very low in the overall adoption (47%) but also only 43% consider it relevant** or applicable to their system. With cloud computing providing the infrastructure to deploy services dynamically [BCS15], we are surprised by this finding;
- **automated recovery (72%) and orchestration manager (71%) rank very high on the respondent's adoption intent.** As cloud systems evolve, managing the infrastructure and services that compose them manually becomes unmanageable. ORCHESTRATION MANAGER facilitates this task, with tools providing embedded AUTOMATED RECOVERY, greatly easying the orchestration effort of the application.

11.3.4 Pattern Relationships

P(column|line)	containerization	external monitor	log aggregation	preemptive logging	job scheduler	automated recovery	failure injection	infrastructure as code	automated scalability	orchestration manager	service discovery	messaging system
CONTAINERIZATION		.64	.81	.45	.64	.52	.20	.78	.74	.59	.59	.74
EXTERNAL MONITOR	.75		.80	.41	.66	.58	.20	.78	.73	.59	.54	.75
LOG AGGREGATION	.76	.64		.55	.68	.57	.23	.81	.72	.62	.57	.80
PREEMPTIVE LOGGING	.69	.53	.91		.71	.64	.31	.71	.69	.56	.58	.82
JOB SCHEDULER	.72	.64	.82	.52		.61	.26	.74	.74	.59	.56	.85
AUTOMATED RECOVERY	.73	.69	.86	.59	.76		.31	.82	.80	.67	.59	.84
FAILURE INJECTION	.78	.67	.94	.78	.89	.83		.94	.78	.72	.78	.94
INFRASTRUCTURE AS CODE	.76	.65	.85	.45	.63	.56	.24		.76	.63	.58	.82
AUTOMATED SCALABILITY	.76	.64	.79	.46	.67	.58	.21	.81		.63	.58	.75
ORCHESTRATION MANAGER	.82	.70	.92	.50	.72	.66	.26	.90	.84		.66	.78
SERVICE DISCOVERY	.85	.67	.88	.54	.71	.60	.29	.85	.81	.69		.92
MESSAGING SYSTEM	.70	.60	.81	.51	.71	.56	.23	.79	.68	.53	.60	

Table 11.3: The conditional probability table, showing the likelihood of adopting the pattern in the column, considering that the pattern in the line is known to have been adopted. For example, line 2, column 1 represents the probability of CONTAINERIZATION having been adopted, considering that EXTERNAL MONITOR was, or P(CONTAINERIZATION | EXTERNAL MONITOR). We highlight the values with probability equal or above 0.8.

Table 11.3 (p. 203) shows the conditional probability[2] between every two patterns in the language, that is, the likelihood of one pattern to be adopted, given that the other is adopted, $P(X|Y)$. This information hints us on pattern dependencies and patterns that are often implemented together. Considering that a probability of *80% or more* means that it is *very likely* for the pattern X to be implemented given that Y is, and that a probability of *60 to 80%* means that it is *likely*, we observe that:

- **log aggregation and messaging system are *very likely* to be implemented with other patterns**, with 79% and 74% probability, respectively. This is not surprising *per se*, given that these two patterns are the most adopted overall;

[2] We are specifically looking beyond joint probability, to assess potential asymmetries in pattern adoption, that is, situations where $P(X|Y)$ differs significantly from $P(Y|X)$.

- **Those using orchestration manager are very likely also to be using containerization (82%), but the opposite is not true (59%).** When using CONTAINERIZATION, one has multiple strategies to deploy and operate the containers, not necessarily requiring an ORCHESTRATION MANAGER. On the other hand, most ORCHESTRATION MANAGER's incentivize the use of containers to run the software, hence, the increased likelihood of adopting CONTAINERIZATION with ORCHESTRATION MANAGER;

- **Those implementing failure injection are likely to also implement other patterns in their systems ($p > 67\%$).** We believe that either (a) FAILURE INJECTION requires a sophisticated system in place, or (b) that developers give it a very low priority. We should note that this conditional probability is highly asymmetric, as discussed in the next point;

- **No other pattern seems to increase the likelihood of adopting failure injection** significantly. FAILURE INJECTION evidences the lowest likelihood of being adopted given any other patterns; we interpret this finding as a kind of *isolation/independence* in the engineering decision of pursuing this pattern concerning the rest of the language.

We are now able to draft a new pattern map (Figure 11.3 (p. 205)) that highlights the relationships uncovered in Table 11.3 (p. 203), compare it to the original in Figure 6.1 (p. 72), and observe that:

- **The indegrees of log aggregation (10), infrastructure as code (6) and messaging system (7) are very high** when compared to the average (2.6). This might be due to their being the top used patterns ($> 69\%$); however, CONTAINERIZATION, which follows immediately on the rank (67%), exhibits a much lower indegree (2).

- **Only 50% of the original connections (8 out of 16) match** those there were inferred;

- **log aggregation and messaging system present a bidirectional relationship,** which might be interpreted as being most likely used in tandem. **The same can be observed with log aggregation and infrastructure as code**;

- **external monitor exhibits the lowest degree** with just one outbound edge.

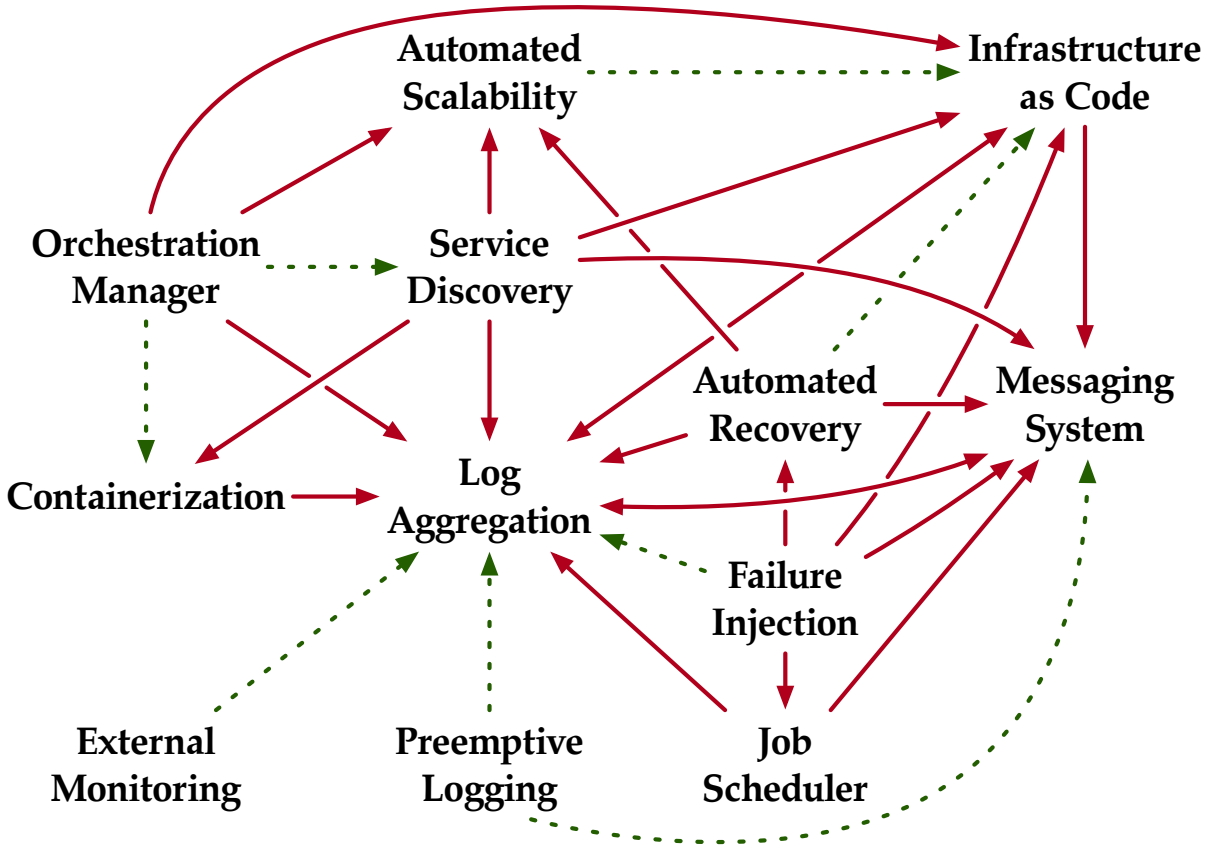


Figure 11.3: Pattern map inferred from the conditional probabilities in Table 11.3 (p. 203), where if $P(B|A) \geq 0.8$, then $A \rightarrow B$. Green dashed arrows match those originally identified. Red solid arrows depict uncovered relationships.

11.3.5 Product Operation Strategy

We expected to observe a correlation between the complexity of the operation strategy and the mean number of patterns adopted. We measure this by averaging the number of patterns that each respondent implemented per operation strategy. We were able to draw the following conclusions:

- **There is no statistical evidence of a relationship between product operation strategy and pattern adoption.** There is an increase in average pattern adoption with the product operation strategy complexity, with means of 5.9, 6.7, and 7.2, as detailed in Table 11.4 (p. 206). We hypothesized that there might be a statistically-significant variation between these populations, either pair-wise, one versus all others, or in their total variance. Table 11.5 (p. 207) presents the t-test and ANOVA statistical analysis for these tests, from which we can conclude that there is no statistical significance ($p < 0.05$) in the variations observed;

- **With OSPC, containerization (85%) and log aggregation (80%) are the**

	Product operation strategy		
	SSSC	SSMC	OSPC
Respondent count	10	72	20
Mean pattern adoption	5.9 ± 3	6.7 ± 3.1	7.2 ± 3.3
Pattern name	**Adoption per pattern (%)**		
LOG AGGREGATION	60.0 ▼	72.2	80.0 ▲
MESSAGING SYSTEM	60.0 ▼	73.6 ▲	70.0
INFRASTRUCTURE AS CODE	60.0 ▼	69.4	75.0 ▲
CONTAINERIZATION	60.0 ▼	63.9	85.0 ▲
AUTOMATED SCALABILITY	60.0 ▼	68.1 ▲	60.0 ▼
JOB SCHEDULER	50.0 ▼	59.7	65.0 ▲
EXTERNAL MONITOR	50.0 ▼	59.7 ▲	55.0
ORCHESTRATION MANAGER	30.0 ▼	47.2	65.0 ▲
AUTOMATED RECOVERY	50.0	45.8 ▼	55.0 ▲
SERVICE DISCOVERY	40.0 ▼	43.1	65.0 ▲
PREEMPTIVE LOGGING	50.0 ▲	44.4	40.0 ▼
FAILURE INJECTION	20.0 ▲	19.4	10.0 ▼

Table 11.4: The first lines of the table show the average number of patterns adopted for each operating strategy, the observed standard deviation and the respective number of respondents with each strategy. On the following lines, we present the percentage of respondents who have adopted each pattern, aggregated by their operating strategy. The ▲ and ▼ icons identify which groups are the highest and lowest adopters.

highest adopted patterns, with CONTAINERIZATION being the overall most adopted pattern with a 25% adoption increase from SSSC. We can theorize that containers are essential for automating service deployment and more relevant with more complex operation strategies. LOG AGGREGATION showed 80% adoption with OSPC, as it streamlines the evaluation of the state of any service, despite the customer where it executes. Deploying several systems presents increased complexity when compared to single deployments, at least due to the number of deployments the team needs to operate, which increases the probability of failure proportional to the number of systems maintained;

- **failure injection is less adopted with OSPC (10%) and most adopted with SSSC (20%).** Applying this pattern is by itself challenging, as it requires very sophisticated automation in place, as well as an increased level of redundancy to prevent unexpected failures or downtime. While this is hard to do for a single system, it is even more complex for OSPC, given that failures could be introduced in more than one system simultaneously, leaving the team overwhelmed. Also, the required investment in redundancy can be prohibitively expensive, as each client will

	Product operation strategy				
	SSSC	SSMC	OSPC	vs. Others	ANOVA
SSSC	—	0.46	0.28	0.39	
SSMC	—	—	0.46	0.84	0.53
OSPC	—	—	—	0.38	

Table 11.5: Evaluation of the significant difference between the three populations. A t-test is applied between every two populations. The *vs. Others* column compares the population against the combination of all other strategies. For all tests, the probability value is shown.

require his system to be scaled beyond its base requirements to recover seamlessly from failures;

- **orchestration manager is adopted ≈2x more often in OSPC than in SSSC.** Applications built specifically for a customer are likely to be delivered in their hardware and changed less often. When working for a variable number of users, with SSMC or OSPC, the need to scale the system is latent. An ORCHESTRATION MANAGER abstracts the underlying infrastructure and the effort to start/stop new service instances;

- **71% of the respondents are using SSMC.** Business to consumer web applications adopts a SSMC operation strategy, allowing the user to register and use a service on demand. We observed that most of the respondents adopted this operation strategy, highlighting its prominence in cloud computing.

11.3.6 Active Monthly Users

We theorize that, as the number of active monthly users increase, so will the number of patterns adopted to cope with the operation of the required infrastructure. The average pattern adoption for each interval is presented in Table 11.6 (p. 208). We could draw the following conclusions:

- **The number of active monthly users grows in tandem with the average pattern adoption, starting at 4.8 with $<$ *100* monthly users and reaching nine at $>$ *1M* monthly users.** The *10k – 100k* bucket was the only exception, arguably due to a non-representative sample. Considering the 4.1 standard deviation observed, we exclude this bucket from the following discussion as an outlier. We hypothesized that there might be a statistically-significant variation over these averages. We evaluated the differences between each group pair-wise, each group compared to all

	Active monthly users					
	$\vee$ 100	100 – 1k	1k – 10k	10k – 100k	100k – 1M	$\wedge$ 1M
Respondent count	18	18	21	16	21	8
Mean pattern adoption	4.8 ± 3.1	6.4 ± 2.6	7.4 ± 2.5	5.8 ± 4.1	7.7 ± 2.2	9.0 ± 3.2
Pattern name	**Adoption per pattern (%)**					
LOG AGGREGATION	38.9 ▼	77.8	90.5 ▲	56.2	85.7	87.5
MESSAGING SYSTEM	55.6 ▼	66.7	61.9	75.0	90.5 ▲	87.5
INFRASTRUCTURE AS CODE	66.7	72.2	61.9	50.0 ▼	85.7	87.5 ▲
CONTAINERIZATION	66.7	66.7	76.2 ▲	50.0 ▼	71.4	75.0
AUTOMATED SCALABILITY	55.6 ▼	66.7	76.2	62.5	57.1	87.5 ▲
JOB SCHEDULER	44.4 ▼	50.0	71.4	50.0	71.4	75.0 ▲
EXTERNAL MONITOR	27.8 ▼	88.9 ▲	71.4	37.5	57.1	62.5
ORCHESTRATION MANAGER	27.8 ▼	44.4	61.9	43.8	47.6	87.5 ▲
AUTOMATED RECOVERY	22.2 ▼	33.3	52.4	50.0	66.7	75.0 ▲
SERVICE DISCOVERY	44.4	33.3 ▼	57.1	50.0	42.9	62.5 ▲
PREEMPTIVE LOGGING	22.2 ▼	33.3	47.6	43.8	57.1	75.0 ▲
FAILURE INJECTION	11.1	11.1	14.3	06.2 ▼	33.3	37.5 ▲

Table 11.6: The first lines of the table show the average number of patterns adopted for each interval of monthly active users, the observed standard deviation and the respective number of respondents with each user interval. The next lines show the percentage of respondents who have adopted each pattern, aggregated by the number of monthly users. We use "k" to represent thousands and "M" for millions. The ▲ and ▼ icons identify the population which adopts each pattern the most and least, respectively.

others, and all groups using a *t-test* and ANOVA. We observed that the variation of the average pattern adoption is meaningful ($p < 0.05$) between the $<$ *100* group and the *1k – 10k*, *100k – 1M*, and $>$ *1M* groups. The $>$ *1M* group also shows a significative difference to the *100 – 1k* group. When compared to others, $<$ *100* and $>$ *1M* groups show significative variation. Such is also observed when the variance of all groups is tested with the ANOVA. This leads us to conclude that the number of active monthly users is a relevant driver of pattern adoption;

- **Companies operating over 1M monthly active users adopt the most patterns.** Other than FAILURE INJECTION, all patterns had a minimum adoption rate of 62.5%. At this scale, it becomes very troublesome to operate cloud software manually, which motivates the 87.5% adoption of ORCHESTRATION MANAGER,

	Active monthly users							
	$\vee$ 100	100 – 1k	1k – 10k	10k – 100k	100k – 1M	$\wedge$ 1M	vs. Others	ANOVA
$<$ 100	—	0.10	$<$ 0.01	0.46	$<$ 0.01	$<$ 0.01	$<$ 0.01	$<$ 0.01
100 – 1k	—	—	0.23	0.55	0.12	0.04	0.70	
1k – 10k	—	—	—	0.13	0.74	0.17	0.23	
10k – 100k	—	—	—	—	0.08	0.06	0.18	
100k – 1M	—	—	—	—	—	0.21	0.11	
$>$ 1M	—	—	—	—	—	—	0.03	

Table 11.7: Evaluation of the significant difference between the active users' populations. A t-test is applied between every two populations. The *vs. Others* column compares the population against the combination of all other strategies. For all tests, the probability value is shown.

AUTOMATED SCALABILITY, INFRASTRUCTURE AS CODE, MESSAGING SYSTEM, and LOG AGGREGATION;

- **Companies operating less than 100 monthly users show high adoption of infrastructure as code and containerization.** Small companies tend to have an increased pressure to optimize their operations, due to limited human resources. Alongside with it, these are typically companies that are starting and so design their system to be cloud-native;

- **There is a smaller representativity of companies with over 1M active users.** Managing over 1M active monthly presents a scaling challenge. We theorize that this group presents a smaller number of respondents as it is the hardest to achieve, requiring sustained success in retention and user acquisition;

- **The *10k – 100k* population might not be representative**, with an average of 5.8 patterns adopted, the group is second to last in average pattern adoption, being surpassed only by the $<$ *100* group. The standard deviation of 4.1, with only 16 responses, suggests an unbalanced group, possibly biased by some respondents. We can confirm this hypothesis by looking into the individual responses from this group. We can see that one respondent responded with 0 patterns, another with one pattern and four with two patterns implemented, effectively biasing the group;

- **failure injection has the highest adoption with companies with $>$ *1M* active users**. We have discussed how few companies have the necessary resources to implement

FAILURE INJECTION. It then makes sense that the companies operating the higher number of users will likely have the highest operation budget and be more capable of implementing the pattern, which justifies why the population managing over 1M active users claim to implement the pattern 37.5% of the times.

11.3.7 Company Size

	Company size			
	Micro	**Small**	**Medium**	**Large**
Respondent count	27	28	17	30
Mean pattern adoption	6.3 ± 3	6.4 ± 3.2	6.6 ± 2.5	7.4 ± 3.4
Pattern Name	**Adoption per pattern (%)**			
LOG AGGREGATION	63.0 ▼	67.9	76.5	83.3 ▲
MESSAGING SYSTEM	63.0 ▼	67.9	76.5	80.0 ▲
INFRASTRUCTURE AS CODE	74.1 ▲	67.9	58.8 ▼	73.3
CONTAINERIZATION	51.9 ▼	78.6 ▲	64.7	73.3
AUTOMATED SCALABILITY	77.8 ▲	60.7	52.9 ▼	66.7
JOB SCHEDULER	55.6	53.6 ▼	76.5 ▲	60.0
EXTERNAL MONITOR	63.0	64.3 ▲	47.1 ▼	53.3
ORCHESTRATION MANAGER	44.4	42.9 ▼	52.9	56.7 ▲
AUTOMATED RECOVERY	37.0 ▼	46.4	52.9	56.7 ▲
SERVICE DISCOVERY	44.4	35.7 ▼	47.1	60.0 ▲
PREEMPTIVE LOGGING	37.0 ▼	46.4	47.1 ▲	46.7
FAILURE INJECTION	14.8	10.7 ▼	11.8	30.0 ▲

Table 11.8: The first three lines of the table show the average number of patterns adopted for each company size, the observed standard deviation and the respective number of respondents. Percentage of respondents who have adopted each pattern, aggregated by the company size, according to the European Commission definition [Eur15]. The ▲ and ▼ icons identify the population which adopts each pattern the most and least, respectively.

As larger companies have more resources available, we expected to observe a correlation between company size and the number of patterns implemented in the company. The results are detailed in Table 11.8 (p. 210), from where we observe:

- **There is no statistical evidence of a relationship between company size and pattern adoption**. There is an increase in average pattern adoption along with the increase in company size, means of 6.3, 6.4, 6.6, and 7.4, as detailed in Table 11.8 (p. 210). We hypothesized that there might be a statistically-significant variation between these populations, either pair-wise, one versus all others, or in their total

	Company size					
	Micro	**Small**	**Medium**	**Large**	**vs. others**	**ANOVA**
Micro	—	0.84	0.66	0.19	0.39	0.52
Small	—	—	0.81	0.27	0.58	
Medium	—	—	—	0.44	0.93	
Large	—	—	—	—	0.15	

Table 11.9: Evaluation of the significant difference between company sizes. A t-test is applied between every two populations. The *vs. Others* column compares the population against the combination of all other strategies. For all tests, the probability value is shown.

variance. Table 11.9 (p. 211) presents the results of the t-test and ANOVA, from which we can conclude that there is no statistical significance ($p < 0.05$) in the variations observed;

- **Micro companies lead infrastructure as code and automated scalability implementation.** When looking at the pattern adoption per company size, we can observe that INFRASTRUCTURE AS CODE and AUTOMATED SCALABILITY both show a higher percentage at both the micro and large companies, with a slightly less adoption in small and medium companies. Small companies have limited human resources, a driver for automating operations. On the other hand, large companies have larger teams and can cope with less automation, but tend to have expertise in house, resources, and time to invest in automation. Time and resources that might not be available with small and medium companies;

- **Medium-sized companies have less representation,** about 42% less than the micro and small company groups. We believe this may be a depiction of reality. According to Eurostat, the number of persons employed for *micro*, *small*, *medium-sized*, and *large* enterprises was 29%, 20%, 17% and 34%, respectively [18].

- **There is a correlation between company size and pattern adoption.** The result shows that the variation between micro, small, and medium companies consist of a slight increase in the number of patterns adopted, ranging from 6.3 to 6.6. The increment to large companies is higher, with large companies implementing an average of 7.4 patterns. Small companies implement over six patterns, which let us theorize that many of these patterns are essential to companies of any size. We can conclude that a correlation is observed between company size and the number of patterns adopted.

- **Company size may mislead for evaluating engineering expertise.** The number of employees is a good indicator of the resources available in the company, but might not translate engineering expertise, as it does not reflect how many employees are working on the product, how many of those are engineers, or how much experience that engineering team shares. In retrospect, the number of engineers in the team would likely provide a better relationship between the resources available to build the product and the pattern adopted. Therefore, we propose that replications of this study should replace this question by two others — *How many engineers are working on the product?* and *How much accumulated experience do they share in years as software engineers?*

11.4 Discussion

This survey sheds light on how the industry adopts the pattern language described in this research, from the personal experience of 102 professionals.

We have reached new conclusions for each of the following questions:

RQ2. What strategies are adopted for addressing cloud problems? While we do not evaluate alternative strategies, we can assert that the respondents adopt all the solutions described in the pattern language to some extent. LOG AGGREGATION is the most adopted pattern, with an average adoption of 72%. FAILURE INJECTION is consistently the least adopted pattern with a 17% adoption rate and is a statistical anomaly when compared to the average adoption rate from other patterns. Its limited adoption is likely due to the redundancy it requires, making it prohibitively expensive for most teams.

RQ4. Are companies that develop software for the cloud aware of these problems and adopt the identified solution? Considering the positive pattern adoption percentage, we can conclude that developers are aware of these problems, and many already implemented some of our patterns to solve them.

RQ5. What characteristics influence the emergence of specific problems when developing software for the cloud? The survey characterized the respondent's company using three variables: product operation strategy, company size, and monthly active users. The collected data demonstrates that the mean pattern adoption increases with company maturity for the three variables studied. The sole exception was for companies with *10k – 100k* active monthly users, possibly as a

result of a non-representative sample in this population. While there is a noticeable increase for all variables, only with the active monthly users can we observe a statistically significant variation between the mean pattern adoption, specifically with the < 100 and $> 1M$ groups. We can conclude that the most influencing company characteristic is the number of active monthly users. We believe that this is the best driver for growth, hence, for inducing the scalability requirements that motivate the adoption of the patterns in our language.

11.5 Threats to Validity

Other works before ours have sought to validate patterns empirically [Sal00; TKP04; Rol+00], but empirical research methods are not without liabilities that might limit the extent to which we can answer our research questions. We identified the following threats to the validity of the conclusions and discussed their impact on this research.

11.5.1 Construct Validity

Using a questionnaire is a common strategy to gather data from a broader audience, but practical considerations tend to limit the type of data captured to be **closed responses**. We could mitigate this by resourcing to other methods in the future, *e.g.* allowing arbitrary text in the response form or performing direct interviews. That would enable a deeper understanding of each approach and particularities, with the added challenge of scaling the answers. Nonetheless, because each strategy has its specific advantages, this research contributes to a future *meta-analysis* that could empower the statistical significance of our findings.

11.5.2 Internal Validity

It is possible for the responses to be inaccurate due to **poor question wording** or **unclear language** [MBV06]. We tried to minimize this threat by accompanying questions with a description and rationale, including relatable examples whenever possible. The questionnaire was disseminated online through social networks, direct contacts, and forums. No measure was taken to verify the respondents' identity, background, or the **truthfulness** of their responses.

11.5.3 External validity

Coverage error, refers to the mismatch between the *target* population and the *frame* population [Cou00]. In this scenario, our target population was cloud professionals, while the frame population was all users who gained access to the link to the questionnaire. We did not take any measures to verify the identity of the respondents. Therefore, some might not possess the profile and knowledge needed to respond to the questionnaire. We have included an introductory text in the questionnaire to filter out individuals not belonging to our target population.

Sampling errors are possible, given that our sample is a small subset of the target population. Repeating the experience would necessarily cover a different sample population, and likely attain different results [Cou00]. Assuming that responses follow a normal distribution about its mean, a large-enough sample can mitigate this threat, which can be evaluated by seeing if 95% of the responses are within two standard deviations from the mean.

Nonresponse errors are introduced by not receiving responses from the chosen target population [Cou00]. Although we disseminated this survey through specific channels, we cannot assess how many of these participated in the survey, so this is an unmitigated risk. As the respondents were volunteers, there might be a bias towards those who have an **affinity with the topic**. Professionals without interest in the subject would be less likely to respond, as they might feel that they had nothing to contribute.

Multiple responses for the same project, which might result from different team members answering the questionnaire, which would introduce errors in our analysis. We mostly discard this threat by making sure that there are no repeated companies in the answers.

Measurement errors refers to the deviation in a response from its actual value. It can result from a lack of motivation, comprehension problems, or intent to damage the study [Cou00] deliberately. To mitigate this risk, we made the questionnaire as short and as clear as possible, so that any volunteer would reply to the questionnaire in a few minutes while preventing partial submissions.

The expertise of engineers influences their cloud approach. The personal experience might be derived from past projects or higher education. Similarly, **incorrect project perception** would have lead to incorrect responses due to a misunderstanding of how their system was designed. Given that we have not captured the respondents' expertise, we cannot evaluate how both threats impact our data.

The **product's start date** can influence its cloud approach. A product started before

the cloud computing era may still make use of processes and practices that are not aligned with cloud computing. We would like to capture this metric in future iterations of our survey.

11.6 Conclusion

The goal of this survey is to shed some light on how the industry adopts the pattern language for developing software for the cloud described in Chapter 6 (p. 69). It does so based on the experience of 102 professionals and by seeking answers to three research questions:

11.7 Summary

This chapter described a survey complementing the validation of our pattern language, initially described in the case study from Chapter 10 (p. 149) by reaching a broader audience and understand how they implement the pattern language.

The data analysis demonstrates that, despite small, there is a correlation between the number of patterns adopted and the product operation strategy, number of active users, or company size. We can use these correlations as an indicator of how relevant these patterns are for maturing the cloud practices of cloud professionals.

During this chapter, we address the following Research Questions (RQs):

RQ2. What strategies are adopted for addressing cloud problems?

While we do not evaluate alternative strategies, we can assert that the respondents adopt all the solutions described in the pattern language to some extent

Are companies that develop software for the cloud aware of these problems and adopt the identified solution?

We observe that the overall adoption of all patterns is 55% and that most companies implement at least one pattern, with only 2 out of the 102 not implementing any pattern.

What characteristics influence the emergence of specific problems when developing software for the cloud?

We survey 102 professionals and evaluate three company characteristics along with their pattern adoption: product operation strategy, active monthly users, and company size. For all three, we observe that there is an increase in mean pattern

adoption as companies mature, except for the *10k – 100k* interval of active monthly users, likely due to a bad sample. This observation is statistically significant with the active monthly users variable, mainly with the < 100 and $> 1M$ populations. The active monthly users were the most influencing metric while evaluating the number of patterns adopted, which allows us to conclude that the number of monthly users influences the scalability requirements and motivate professionals to adopt new patterns from our language.

While the results observed were aligned with our hypothesis, the volume of responses and the queried population sample might not represent the global population of cloud professionals. Future work should improve the number of responses to improve the results' quality and correlation confidence. A new study could also try to understand the motives for the variations in pattern adoption in the different analyzed intervals. A larger-scale interview could provide relevant data for such an assessment.

Also, it would be relevant to analyze this data from an operations research standpoint to identify the ideal order in which patterns should be adopted, creating a detailed adoption guide for the pattern language. Such a study could benefit from company characterization and identify optimal pattern implementation sequences for different company characteristics.

Chapter 12

Conclusion

The widespread of the Internet-enabled unprecedented growth in the number of applications achieving global reach. Cloud computing provided the technology to facilitate building such applications. With the paradigm, along came the need for new architectures, practices, and tools. With this work, we contribute to the field of Software Engineering, specifically to the domain of cloud computing with a pattern language of twelve patterns, ten of which novel from this work. We then validate the adoption of these patterns in the industry employing a case study with five local startups and a survey with over 100 industry professionals. As a direct or indirect result of this research, we have authored 18 papers for peer-reviewed conferences, listed in Appendix C (p. 255).

12.1 Research Questions

The information made available for supporting cloud application design is often a result of personal experience and limited observation, biased to specific application contexts, lacking strong scientific support, and adaptability to specific contexts. That might be the reason why still in 2019, the lack of cloud expertise is one of the most critical challenges for cloud development [Rig19].

This work was driven by five research questions directly related to our hypothesis. We can conclude that:

RQ1. What are the recurrent problems when developing software for the cloud?

Cloud applications introduce development challenges, viz., (1) being able to scale, (2) coping with dynamic infrastructure and service orchestration, (3) discovery, (4) monitoring, (5) isolation, (6) messaging, (7) availability, (8) reliability, (9) resiliency, and (10) security (see Chapter 3). While cloud development has been around since 2006, addressing these concerns continues to be challenging, with lack of quality resources and expertise being the most severe limitation in cloud development. In this dissertation we focus on 10 recurrent problems (P) in cloud computing (cf. Chapters 6 to 9). There are obviously tens to hundreds of other relevant recurrent problems while designing software for the cloud, which could be researched in future iterations of this work. In this context, we address the following:

P1. Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself;

P2. Manually operating software at scale, particularly in architectures that favor microservices and their cooperation, is an error-prone, slow and costly process;

P3. Services will eventually fail in the long run and need to be recovered in a timely and orderly fashion;

P4. Short-running jobs need to be scheduled and orchestrated using dynamic infrastructure without permanently allocating resources, possibly requiring ephemeral hardware to execute;

P5. Resilience mechanisms are triggered when the software is failing. Since systems are designed to work correctly, the *status quo* resists to a continuous verification of the correctness of those mechanisms. To ensure resilience, we need to exercise failures to evaluate their impact;

P6. The information required to debug failures is often lost during their first occurrence due to insufficient log verbosity;

P7. Services orchestrated at scale produce widely disperse information, resulting in a complicated process to navigate and correlate multiple sources;

P8. Monitoring an application from its inner layers results in an incomplete or biased version of the reality;

P9. In a dynamically allocated infrastructure, services require a discovery strategy to establish a communication channel;

P10. As the volume and complexity of interacting services increase, point-to-point communication channels become unmanageable, hindering fault-tolerance, resiliency, and scalability.

RQ2. What strategies are adopted for addressing cloud problems?

As developers attempt to address the aforementioned problems, recurrent solutions began to emerge. The following solutions, that we captured in pattern form, address those problems:

Containerization. Use a container to package the service and its dependencies and enable its isolated programmatic deployment;

Orchestration Manager. Adopt an ORCHESTRATION MANAGER to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements;

Automated Recovery. Include checks and recovery strategies in the instructions provided to the ORCHESTRATION MANAGER to orchestrate containers, enabling it to monitor and recover failing containers;

Job Scheduler. Deploy a scheduler service along with the ORCHESTRATION MANAGER that can instruct it to allocate *one time* or *periodic jobs*, releasing their resources for reuse in the cluster when they complete;

Failure Injection. Generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms to verify the application's resilience;

Preemptive Logging. Adjust logging verbosity in services and servers within acceptable resource limits, maximizing the probability of capturing relevant information for addressing future issues right from their first occurrence;

Log Aggregation. Aggregate and index all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs;

External Monitor. Test the application's public interfaces from an external source, increasing the confidence over the application's status;

Service Discovery. Abstract service network details by relying on an external mechanism that facilitates communication and balances traffic between two services;

Messaging System. Use a MESSAGING SYSTEM, colloquially known as *message queue*, to abstract service placement and orchestrate messages with the optimal routing strategy between them.

RQ3. What driving forces influence how strategies are implemented?

Each problem described is frequent in the context of cloud development. Nevertheless, the context under which the problem is observed can influence how the solution needs to be curated to adjust to the specific case. Each problem will have its variants that need to be considered and adapted while implementing the solution. Our pattern language introduces a list of forces for each pattern, describing how the problem can vary. Solutions can often be adjusted based on these forces for a better fit in their context. The solutions to our patterns take into consideration several forces, some of which being (1) automation, (2) decoupling, (3) isolation, (4) latency, (5) portability, (6) reliability, (7) resilience, (8) resource allocation, (9) scalability, (10) security, and (11) supervision, cf. Chapters 6 to 9 (pp. 69, 77, 117 and 135).

RQ4. Are companies that develop software for the cloud aware of these problems and adopt the identified solution?

Patterns, by definition, are observations of the strategies applied by developers to recurring problems. Such does not mean they capture (1) *the best* practice for a problem, nor that (2) professionals are aware and use those practices. To validate our pattern language's relevance, we produce a case study by interviewing five companies and evaluating how they are designing their cloud software. These interviews let us understand which patterns they use and how. Companies that operate multiple independent instances of their product implemented all but one pattern from our language. They highlight that automation is essential to prevent human error and make operations efficient. In contrast, the other companies implement at most three patterns from the language, intending to adopt two more patterns in the near feature. We ask the five companies what would be needed to scale their application considerably, and those that adopted most patterns considered scaling to be trivial to scale, given the required computational resources. The others considered they would have to change their design, often referring to the need to implement new patterns to cope with the new scale. The case study enabled us to iterate some of

our patterns with new forces and implementation details that we have not previously observed, cf. Chapter 10 (p. 149).

In addition to the case study, we also surveyed 102 industry professionals who responded if they have implemented each pattern into their product, cf. Chapter 11 (p. 191). We learned that 98% of the professionals adopt at least one pattern and that the mean pattern adoption was of 56% ± 15.82, meaning that any given professional is likely to implement a specific pattern 56% of the times. While evaluating the average pattern adoption considering company and product characteristics, namely operation strategy, active monthly users, and company size, we have observed an increase in the average number of pattern adoption with the increase in maturity for each variable. When hypothesized if there was a strong relationship between maturity for each variable and the increase in pattern adoption, we only found statistical significance with the number of active monthly users, which allow us to conclude that user volume is the critical driver for the appearance of the problems identified and the adoption of the related pattern.

RQ5. What characteristics influence the emergence of specific problems when developing software for the cloud?

The pattern language addresses problems that might emerge at different stages for different companies. To understand if and what company characteristics influenced the adoption of specific patterns, we surveyed over 100 professionals, cf. Chapter 11 (p. 191). Along with their pattern adoption, we asked them to classify their company regarding three variables: (1) operation strategy, (2) the number of active monthly users, and (3) company size. We then analyzed the pattern adoption for each of these variables. We learned that, as companies mature, that is, as the operation strategy or volume of active monthly users increases or the company grows, they tend to adopt more patterns. While the mean pattern adoption is increased for all three variables, the difference is *statistically significant* only with the number of active monthly users, mainly in the lowest and largest user bucket.

12.2 Hypothesis Revisited

The Research Questions (RQs) drove this research intending to address our main hypothesis:

> *While engineering software for the cloud, there are categories of recurring problems, which solutions converge from* good design *principles, that adjust*

to the context where they emerge. Their adoption is a consequence of (1) the awareness a team has of a problem, (2) the characteristics of the product and the company, and (3) the way these solutions relate amongst themselves.

The answers to those RQs allow us to deconstruct and discuss how we have addressed the hypothesis:

While engineering software for the cloud, there are categories of recurring problems. We identify twelve of these recurring problems from cloud software development. They are classified into four categories: automated infrastructure management, orchestration and supervision, monitoring, and discovery, and communication.

Solutions to recurring cloud problems converge from *good design* principles. For each recurring problem, we were able to refer or mine a solution strategy in the form of a pattern. The patterns could be observed in the wild with at least three independent implementations. They provide engineers with detailed instructions to solve the identified recurrent problems.

Solutions are adjusted to the context from where they emerge. Solutions mined as patterns are flexible, with a set of forces that can be balanced in a multitude of ways to fit the implementation's context better. The proposed patterns identify such forces and provide implementation details that balance these forces.

The solution's adoption is a consequence of the awareness a team has of a problem. We evaluate how familiar cloud professionals are with the identified problems by performing a case study and a survey, measuring how often the pattern language is adopted in the industry.

The adoption of the solutions is a consequence of the characteristics of the product and company. We identify several relations between the product and company's maturity and the average number of patterns a team adopts. We observed that these relations are particularly relevant to the volume of active monthly users.

The adoption of the solutions is a consequence of the way these relate amongst themselves. In a given context, multiple recurring problems are likely to emerge together. A pattern language goes beyond identifying solutions to independent problems by elaborating on the relationships of the problems and solutions identified. We not only identify the pattern relationships in our language but statistically verify those relationships with our survey.

12.3 Main Contributions

This dissertation contributes to the field of software engineering, and particularly to cloud computing, with (1) a literature review of the state of the art of design patterns for cloud software development, (2) a reference architecture for cloud computing, (3) a pattern catalog, and respective industry case study of cloud and DevOps practices, (4) the pattern language, and (5) the validation composed by a case study with five companies and survey with 102 professionals. Figure 12.1 (p. 223) visually identify the contributions made during these stages towards the domain of Software Engineering, revisited in the remainder of this section.

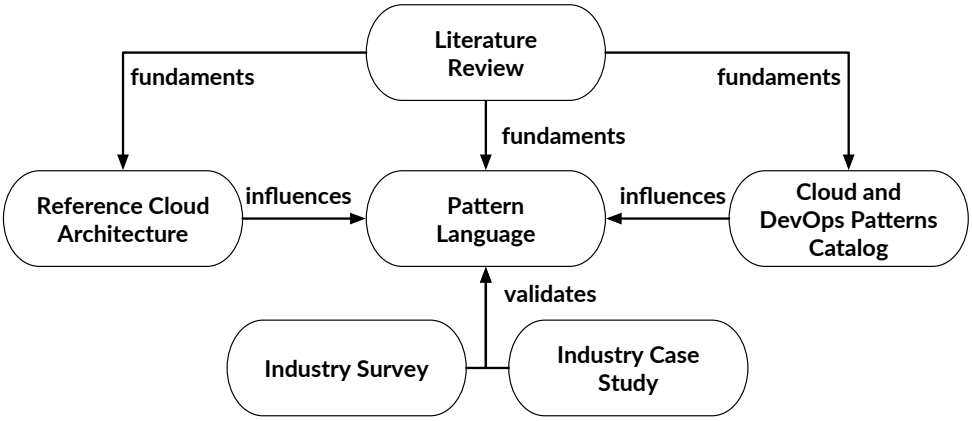


Figure 12.1: Relationship between the contribution items of this research.

12.3.1 Review of the State of the Art

In this work, we revisit the concepts that motivated cloud computing in Chapter 2 (p. 11). Later in Chapter 3 (p. 25), we identify the intricacies of cloud computing, along with the resources available for supporting software development for the cloud. We acknowledge and discuss the recurrent problems for developing for the cloud, asked in RQ1, recognizing that the lack of expertise is the most relevant driver constraining cloud development. This literature review systematizes knowledge regarding cloud design practices and how authors have contributed to it using patterns. It supports the remaining of our research and can be used by other authors to support their work as well.

12.3.2 Reference Cloud Architecture

As researchers and engineers, we considered essential to become experts in the topic of cloud computing, not only theoretically, but with hands-on experience. Chapter 5

(p. 55) describes how we have approached the subject by contributing with a reference architecture for a cloud application of a research project and a case study with Portuguese startups regarding their cloud operations practices and tools.

Section 5.1 (p. 55) described the contribution for a publicly funded research Ambient Assisted Living for All (AAL4ALL), to which we have supplied a reference cloud architecture and prototype to orchestrate the message passing between components in the ecosystem, ensuring scalability, security, and privacy. Despite being implemented considerably before the development of the pattern language, it already applied several of the patterns we would later identify and capture. Part of this contribution proposed a test bench for experimenting with cloud architectures, which culminated in a journal publication, in Appendix C.2.9 (p. 263).

12.3.3 A Pattern Catalog for DevOps and Cloud

Once we had sufficient experience with cloud architectures, tools, and practices, we understood that we needed to learn how the industry was addressing those same challenges. Section 5.2 (p. 62) described a case study where we have interviewed 25 companies developing software, inquiring them about the tools and development practices and their usage for cloud development, enabling us to address RQ2. We created a pattern catalog with 13 patterns. We applied the catalog in an experiment with a local startup, which demonstrated an increase in multiple development efficiency metrics after the team was provided with the pattern catalog and two weeks to implement the patterns.

12.3.4 Patterns and Pattern Language

We captured recurring problems and their solutions in the form of ten novel patterns supported by our previous research. Further literature research and experimentation empowered us to address RQ3 and write these patterns. While trying to answer RQ4, we identified the intricacies of each problem-solution pair as a list of forces. A particular combination of these forces generated a unique configuration of the problem, which required balancing to find a solution fit to the problem.

These ten novel patterns evolved into a pattern language, depicted in Figure 12.2 (p. 225), helping professionals navigate the language by clarifying the most relevant pattern relationships. To further guide the cloud professional to implement these patterns, Section 6.4 (p. 74) describes a particular sequence of adoption for these patterns for implementing a simple web application.

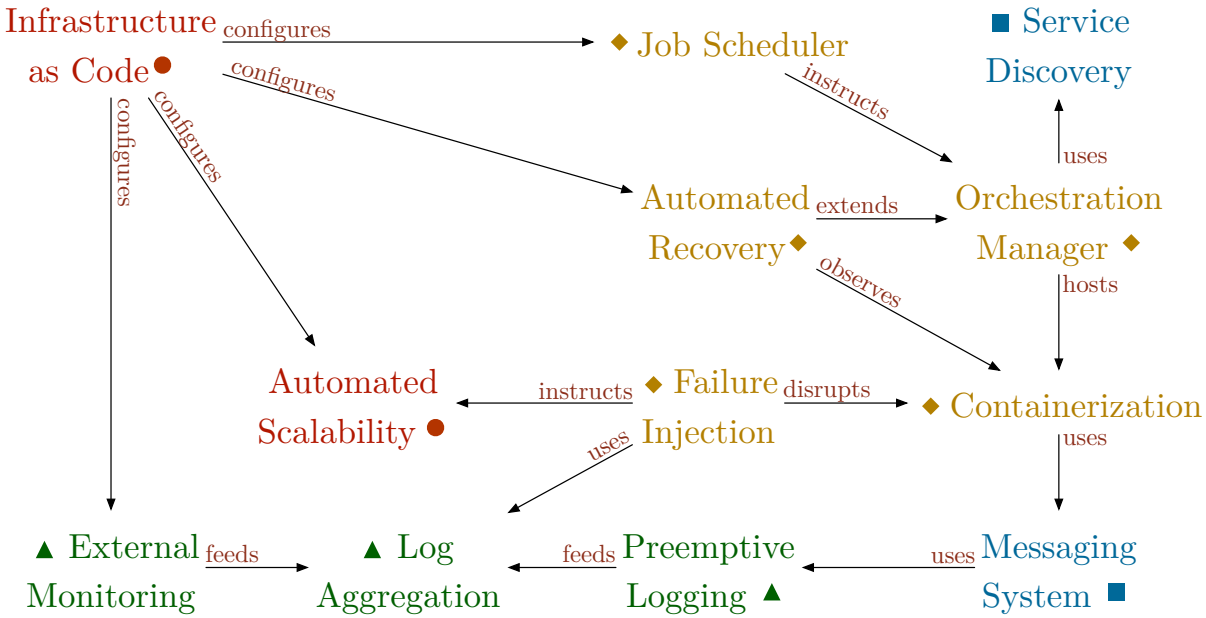


Figure 12.2: Pattern map for the pattern language for engineering software for the cloud, depicting the relations between the patterns. The dashed rectangles limit each pattern category.

12.3.5 Thesis Validation

We have cited arguments in favor of the implicit validation of patterns, which state that they are, by definition, valid, since they are just capturing the reality observed in the wild, being accepted as such once three occurrences are observed. Nevertheless, we can only say that patterns present *a* solution to which some developers converged, not *the best* solution for a given problem. A perfect solution would demand a deterministic context, which is nearly nonexistent in software engineering. We capture the drivers that make each problem unique as pattern forces, which we help balance in our solution description. Capturing forces is a continuous pursuit for the pattern author, as it is also difficult to be sure that all possible drivers are identified.

To verify that our patterns were applicable to the industry, we designed a two-step complementary validation strategy that provided us with additional knowledge for iterating the pattern's specification.

Industrial case study. We started with a case study with five startups to evaluate how they related to the pattern language. We used semi-structured interviews, following a script but motivating the respondents to delve into the intricacies of their systems and design decisions. Using a methodology using open responses enabled us to capture new knowledge that further improved our pattern language. This process was described in Chapter 10 (p. 149) and addresses RQ2, RQ4, and

RQ5. Despite the confined statistical significance, we were able to gather additional knowledge for iterating our pattern language with new forces and more detailed solution descriptions. We observed that companies at different stages have different cloud requirements, and that reflects on their practices and patterns implemented. A study with a larger audience would help identify how company maturity was influencing pattern adoption.

Pattern language adoption survey In Chapter 11 (p. 191), a survey inquired over 100 cloud professionals to understand what patterns they were applying in their cloud products and classify their company maturity in terms of operation strategy, number of active monthly users, and company size. We observed that for the three variables, a correlation exists between the company maturity and the mean number of patterns implemented. We were able to address RQ2, RQ3, and RQ5.

These two validation strategies complemented each other. The interviews provided an understanding of how respondents build their cloud software despite the limited number of interviewed subjects. The survey provided less information regarding the respondent's cloud design, but inquired a larger population, enabling statistically relevant conclusions from the captured data. Together, they allowed to thoroughly understand how companies address cloud development and the product and company characteristics that influence pattern adoption.

12.4 Future Work

We have merely begun the work needed to thoroughly capture the vast cloud design knowledge being applied by professionals for building cloud applications.

Expand the pattern language. This work addresses 12 recurrent problems from engineering software for the cloud. These problems only begin to address the overall set of problems developers have to address while designing their software. We would like to see this language expanded to tens or hundreds of patterns and becoming a reference for engineers. Expanding the language would benefit from the cooperation with other authors, possibly expanding their existing work to the level of maturity that we propose.

Evaluate the solution's strategies. The solutions we capture on the ten novel patterns in our pattern language describe strategies to address the identified

recurrent problems. While we have observed multiple success cases with the application of each one of these strategies, we cannot assert that these are *the best* solutions for the problems. This research would be improved with an investigation for alternative solutions for each problem identified, and an increased discussion on how alternative implementation strategies influence the overall design of the product and balance of the forces.

Pattern improvement Regarding the completeness and correctness of the pattern language, we consider that both can continuously be improved, ideally via additional case studies. Despite interacting with only five companies, we have acquired knowledge regarding cloud approaches, capacitating us to improve and grow the pattern language, making it more accurate, complete, and useful for the pattern adopter. It is then logical that additional case studies would deepen the knowledge of how professionals are addressing the cloud, which would translate into an enrichment of the pattern language with more complete and original patterns. The ideal scenario would be to continue to conduct case studies and improve the language for as long as it will generating new knowledge.

Pattern adoption sequence guideline. Future work could apply observational methods to evaluate how companies developing products from scratch address these challenges. A large enough sample would allow capturing the natural adoption sequence for the patterns in the pattern language, allowing the creation of a concrete adoption guideline.

Improve the survey. The survey presented in Chapter 11 (p. 191) is not without flaws. We concluded that the number of employees in the company is not a relevant metric to evaluate, given that this does not hint on the size and experience of the engineering team working on the product. Repeating the survey asking how many engineers are working on the product and what is their accumulated experience could provide new insights on how engineering expertise can influence pattern adoption.

Pattern Language impact experiment. We would like to evaluate the impact of the pattern language in a controlled experiment, where a first would solve a cloud challenge empowered with the pattern language, while the control group would have to solve the same challenge without the language. We theorize that the first group would outperform the other in terms of development speed and overall solution quality. We expect this study to be complex to execute, as the challenge would be troublesome and require several days to complete. Obtaining a relevant sample of

professionals, or ideally, of teams of professionals, with the availability to commit to the experience, would likely be very difficult without paying those professionals. On a smaller scale, a quasi-experiment could also be designed with volunteers who would have to solve a set of smaller cloud-related problems, with and without the pattern language, evaluating the efficiency and quality that adopting the pattern language could bring to cloud software development in the short-term.

Participatory Observation. We would like to conduct an experiment with participatory observation, similar to the one described at the end of Section 5.2 (p. 62), in which we would provide the pattern language to actual professionals, evaluating how they improve their practices when while applying it. We would evaluate performance metrics from the company before, during, and after the experiment to measure improvements resulting from adopting the pattern language.

Bridge knowledge to other application domains. Cloud computing pioneered the facilitated access to large scale computation, which motivated several new topics within Software Engineering, such as Serverless, Internet of Things, or Fog Computing to emerge. We believe that most concepts introduced in this research apply to those fields as well, while at a different scale, or requiring minor adjustments. We would like to cooperate with researchers from these fields to evaluate if some of these patterns could be expanded or rewritten in the context of their research field.

12.5 Epilogue

Cloud Computing brought exciting changes to how we build software. On the one hand, it provides building blocks to develop very complex applications. On the other hand, it enabled developers to run their applications at an unprecedented scale without a prohibitive initial cost. Together, they empower developers to build applications that reach users on a global scale, making technology an incredibly attractive platform to develop new or improved businesses.

Enabled by cloud computing, technology continues to expand to new domains. Smartphones and Internet of Things (IoT) are becoming ubiquitous, factories and cities becoming smart, and Machine Learning enabling personal assistants to each one of us. With only 14 years since the cloud was introduced, we can only imagine how the future will be. We hope this work can keep expanding and empower future generations of cloud applications.

To those reading this dissertation as inspiration for your own, allow me to share with you the advice I got from a senior professor when I was starting. *If you want to complete your Ph.D., don't change the place where you live, don't get a new girlfriend, don't get a job, don't change a thing, focus on your Ph.D., work hard, you'll have time for everything else once you are done. You will have plenty of time to do everything else later in life.* Well, I failed to follow this advice. All of it. Multiple times. But I was pretty happy at failing them. By doing so, I have learned a lot in life and in the industry along the way. That positively reflects on who I am today, personally and professionally, and in the contents of my Ph.D. So, my advice differs from the one above. To become a *Doctor in Philosophy* is an arduous path. If you are to cross it, find a subject you are passionate about, which you would even research in your free time. Keep your head high during stressful times, and there will be some. Please don't ignore the outside world. You can learn much from it. Work a lot and have fun! If you stay focused, the rest will play out by itself. If you need more help, we wrote a paper that might help [FRS19]!

A final word goes into answering that recurrent uncomfortable question: *have you finished your dissertation yet?* **Yes!**

Appendices

Appendix A

Cloud and DevOps Preliminary Survey

A.1 Interview Protocol

A.1.1 Interview Guide Product (IGP)

IGP1. What type of product do you develop?

IGP2. What is the scale of that product? Number of countries, number of users?

IGP3. Do you have an SLA or some requirements that impact your work?

A.1.2 Teams (T)

T1. How many teams do you have?

T2. What is the size of each team?

T3. Are teams specialized, or do they have multiple specializations working together?

T4. Do teams interact with each other?

T5. How are teams seen from an external perspective? Are they autonomous?

T6. How do you manage your workload? Do you use SCRUM, Kanban, or other?

T7. How do team members communicate among themselves?

A.1.3 Pipeline (P)

P1. How long does your code take to go from idea to production?

P2. What are the states that your code goes through before reaching a production environment?

P3. What triggers the transition between states?

P4. What kind of tests do you develop? Which teams are involved in that process? When do they run?

P5. What happens in each of the pipeline states?

P6. In each state, which teams intervene and what do they do?

P7. What processes did you automate? Did you choose not to automate some? If so, why?

P8. How do you handle your deployment process? Which tools do you use? Do you use containers or Virtual Machines (VMs)?

A.1.4 Infrastructure Management (IM)

IM1. How do you scale? Horizontally or vertically?

IM2. Does scaling happen automatically?

IM3. What can make infrastructure scale up/down?

IM4. How is infrastructure increased?

IM5. How do you lift new instances of your infrastructure? Is it automatic

A.1.5 Monitoring and Error Handling (MEH)

MEH1. What metrics do you collect from running servers?

MEH2. What do you see as errors?

MEH3. What process do you follow to solve errors after they are detected?

MEH4. When errors are detected, who is notified? How is the notification sent?

MEH5. If errors are detected before the software reaches production, what do you do?

A.2 Preliminary Survey Responses

Responses were gathered by individually interviewing 25 companies, mostly based out of Porto and Lisbon, Portugal. Results are described in Table A.1 (p. 236) and Table A.2 (p. 237).

Company Name	Alerting	Auditability	Cloud	Code Review	Communication	Continuous Integration
Abyssal	○	●	●	●	●	●
Arealytics	○	○	●	○	●	○
Atiiv	○	●	●	○	●	○
Nmusic	●	○	●	○	●	●
Zarco	○	●	●	○	●	○
Company A	○	○	●	○	●	●
Celfinet	○	○	●	○	●	○
clickly	○	●	●	○	●	○
Codacy	●	●	●	●	●	●
Emailbidding	●	●	●	●	●	●
EZ4U	○	●	●	○	●	○
Hypelabs	○	○	○	○	●	○
Imaginary Cloud	●	●	●	●	●	●
Iterar	○	●	●	○	●	●
Jscrambler	○	●	●	○	●	●
Mindera	●	●	●	●	●	●
Semasio	○	○	○	●	●	○
Shiftforward	●	●	●	●	●	●
Streambolico	○	○	○	○	●	○
TOPDOX	●	●	●	●	●	●
Ubiwhere	●	●	●	●	●	●
Unbabel	●	●	●	●	●	●
Virtus.ai	○	○	●	○	●	○
3Port	○	○	○	○	●	○
Weduc	○	●	●	●	●	○
Total	9	17	21	11	25	16

Table A.1: Responses to the preliminary survey (1/2).

Company Name	Deployment	Error Handling	Jobs	Reproducible Environments	Scaling	Team Orchestration	Version Control
Abyssal	○	●	○	○	●	●	●
Arealytics	○	○	○	○	○	○	●
Atiiv	●	●	○	●	●	○	●
Nmusic	●	●	○	●	●	●	●
Zarco	○	○	○	○	●	○	●
Company A	●	●	○	●	●	●	●
Celfinet	●	●	○	●	○	●	●
clickly	○	●	○	○	●	○	●
Codacy	●	●	●	●	●	●	●
Emailbidding	●	○	○	●	●	●	●
EZ4U	○	○	●	○	●	○	●
Hypelabs	○	○	○	○	○	○	●
Imaginary Cloud	●	●	○	●	●	○	●
Iterar	○	○	●	○	○	○	●
Jscrambler	○	○	○	●	○	○	●
Mindera	●	●	○	●	●	●	●
Semasio	○	○	○	●	○	○	●
Shiftforward	●	○	○	●	●	○	●
Streambolico	○	○	○	○	○	○	●
TOPDOX	○	○	○	○	○	○	●
Ubiwhere	○	○	○	○	○	●	○
Unbabel	●	●	○	●	○	●	●
Virtus.ai	●	○	○	●	○	○	●
3Port	○	○	○	●	○	○	●
Weduc	○	○	○	●	○	○	●
Total	11	15	3	15	12	9	24

Table A.2: Responses to the preliminary survey (2/2).

Appendix B

Survey

B.1 Questions

The following pages describe the survey disseminated amongst industry professionals to assess their pattern language adoption, detailed in Chapter 11 (p. 191). The following pages include the original questionnaire, while Appendix B.2 (p. 246) describes the individual responses.

Cloud Patterns Adoption Survey

The answers to this questionnaire will create a clear picture of how cloud patterns are adopted throughout software development companies of all sizes. The patterns were captured as part of the Ph.D. work by Tiago Boldt Sousa at the Faculty of Engineering, University of Porto.

If you are working with multiple cloud applications, please consider your main/largest product or the one you are more familiar with for answering these questions.

Please leave your email in the comments if you would like to get further information about this research.

Thanks in advance,
Tiago Boldt Sousa

* Required

1. **Product operation strategy** *
   Consider how you deploy your software, regarding the users that it is intended for.
   Mark only one oval.

   - ◯ One system per customer (Private deployment for each customer, private to his users)
   - ◯ Single system, multiple customers (shared platform for any customer, e.g. Netflix, Facebook)
   - ◯ Single system for single customer (only one deployment for a specific group of users)
   - ◯ Other: _____

2. **Active monthly users** *
   Consider the average number of users for your platform, across all system instances
   Mark only one oval.

   - ◯ < 100
   - ◯ 100 - 1 000
   - ◯ 1 000 - 10 0000
   - ◯ 10 000 - 100 000
   - ◯ 100 000 - 1 000 000
   - ◯ > 1 000 000

3. **Adopted Cloud Providers** *
   Check all that apply.

   - ☐ Amazon Web Services
   - ☐ Google Cloud
   - ☐ Microsoft Azure
   - ☐ Private Infrastructure
   - ☐ Hybrid / Multicloud
   - ☐ Virtual Private Server
   - ☐ Other: _____

4. **Have you adopted Infrastructure as Code?** *

You automate your infrastructure and deployment operations programmatically. Example: Terraform, chef, ansible.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

5. **Have you adopted Automated Scalability?** *

Your system scales dynamically to adjust to elastic traffic. Example: AWS EC2 Auto Scaling.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

6. **Have you adopted Containerization?** *

Deploying a service to a host couples it with the operative system, possibly introducing side effects with other services in the same host, or the host itself. Containerization proposes the usage of containers to package the service and its dependencies and enable its isolated and programmatic deployment. Example: Docker
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

7. **Have you adopted an Orchestration Manager?** *

Deploying and updating software at scale is an error-prone, slow and costly process. Such can be facilitated by adopting an Orchestration Manager to coordinate, manage and distribute multiple cloud services while abstracting the underlying infrastructure, fulfilling the service requirements. Example: Kubernetes, Mesos + Marathon.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

8. **Have you adopted Automated Recovery?** *

Services may fail during execution and need to be recovered in a timely and orderly fashion. Including health checks and recovery configurations in the instructions used for the Orchestration Manager to orchestrate containers, enables it to monitor and recover failing containers. Example: Kubernetes Pod Lifecycle.
Mark only one oval.

- Yes, for the entire system
- Yes, partially
- Under assessment or development
- No, but we would like to
- No, it is not relevant
- I don't know / I don't want to answer

9. **Have you adopted a Job Scheduler?** *

Cloud applications require frequent short-running jobs to be scheduled, which must be orchestrated across a dynamic infrastructure without permanently allocating resources. A scheduler service running along with the Orchestration Manager can instruct it to allocate one time or periodic jobs, recovering their resources to the infrastructure when they complete. Example: Kubernetes Cronjobs.
Mark only one oval.

- Yes, for the entire system
- Yes, partially
- Under assessment or development
- No, but we would like to
- No, it is not relevant
- I don't know / I don't want to answer

10. **Have you adopted Service Discovery?** *

Services might lack the network information required to communicate with other dynamically allocated services. Communication can be achieved by abstracting service network details by relying on an external mechanism that facilitates communication and balances traffic between two services. Example: Kubernetes DNS, Marathon reverse proxy.
Mark only one oval.

- Yes, for the entire system
- Yes, partially
- Under assessment or development
- No, but we would like to
- No, it is not relevant
- I don't know / I don't want to answer

11. **Have you adopted a Messaging System?** *

As service instances increase, communication between services needs to be abstracted, enabling proper balancing between instances. This communication strategy is required to be fault-tolerant and scalable to maintain the application's resiliency. As a solution, a messaging system, colloquially known as message queue, can abstract service placement and orchestrate messages with multiple routing strategies between them. Example: RabbitMQ, Kafka.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

12. **Have you adopted Failure Injection?** *

Resilience mechanisms are triggered when software is failing. Since systems are designed to work correctly, the status quo prevents us to from continuously verifying the correctness of those mechanisms. We need additional strategies to minimize the probability of failure in production due to faulty resilience strategies. Failure injection software can generate atypical events at both the application and infrastructure level, exercising the available recovery mechanisms, verifying the application's resilience. Example: ChaosMonkey from Netflix .
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

13. **Have you adopted Preemptive Logging?** *

The information required to debug issues in software is often lost during their first occurrence due to insufficient log verbosity. By adjusting logging verbosity preemptively in services and servers within acceptable resource limits (CPU, storage, others), the team maximizes the probability of capturing relevant information for addressing future issues right from their first occurrence.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

14. **Have you adopted Log Aggregation?** *

Services orchestrated at scale produce disperse logs, resulting in a troublesome process to acquire and correlate those who come from multiple sources. This pattern suggests the Aggregation and indexing all service and server logs in a central repository, providing the team with a centralized system to query and visualize execution logs. Example: Kibana, GrayLog.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

15. **Have you adopted External Monitoring?** *

Monitoring an application from inside the infrastructure that hosts it will result in an incomplete and biased version of the reality, for example, given the inability to observe issues such as lack of Internet connectivity or abnormal latency to the application. External Monitoring suggest testing the application's public interfaces from an external source, providing an unbiased awareness of the application's status. Example: StatusCake, Pingdom.
Mark only one oval.

- ◯ Yes, for the entire system
- ◯ Yes, partially
- ◯ Under assessment or development
- ◯ No, but we would like to
- ◯ No, it is not relevant
- ◯ I don't know / I don't want to answer

16. **Your role in the product development** *

Mark only one oval.

- ◯ CTO
- ◯ CEO
- ◯ CIO
- ◯ Senior Engineer
- ◯ Engineer
- ◯ Architect
- ◯ Team leader
- ◯ Other: _____

17. **Number of collaborators in the company** *

Mark only one oval.

- ◯ 1 - 10
- ◯ 11 - 50
- ◯ 51 - 250
- ◯ > 251

18. **Company name**

_____

19. **Country**

20. **Comments**

Please let us know if you have any interesting detail to share regarding your infrastructure, application architecture, or this questionnaire. Feel free to share your email if you want to receive further details about this research.

Thanks for your collaboration!

B.2 Responses

We got 102 responses to the survey from companies from all geographic regions, sizes, and levels of maturity. Some respondents shared their place of work, so we know we have reached the following companies: ABC, Agfa Healthcare, B6, Barkyn, be.ubi, BySide, BytePitch, Codavel, Comcast, CompStak, CustomerGauge, Dataform, Desjardins, DigitalOcean, E-goi, Eondeotec, FARO Technologies, Facebook, Feedzai, Hostelworld, InVision, Infraspeak, Insight Software, Lead Forensics, Loqr, Mindera, OLR/E2X, PaddyPowerBetfair, Pigeonlab, Playax, Reach plc, Scalyr, Smarkio, Trinity Mirror, Ultimaker, Utility warehouse, VMuse, Lda, Velocidi, XING, Yahoo!JAPAN, and Yapily.

Responses are individually listed in the tables in this section. Responses are identified with an anonimized sequential ID to hide the relation between the company name and their responses. Table B.1 (p. 248), Table B.2 (p. 249), Table B.3 (p. 250) classify the respondent regarding country, respondent role, product strategy, active monthly users, and company size. Table B.4 (p. 251), Table B.5 (p. 252),and Table B.6 (p. 253), identify which patterns the respondent adopted in their product.

Some respondents provided some comments with their responses. Most were email contacts to receive a follow-up regarding this research. The others were the following:

C1. We are still in a first stage of deploying services to the cloud that support our desktop software. Current services help bridge customers with the software we build (logging, crash detections, updates). We started with Azure Web apps (non containerized, but isolated deployment) and are currently moving to IaC and containerized apps to make the delivery process better and scalable. Mentioned collaborator numbers are only for the Portugal team, as though the company is global, teams work more or less independently.

C2. Hope I didn't break any nda.

C3. You should have a section for serverless, we did a mixture of containers and Lambda for our architecture but I couldn't accurately reflect that given the questions.

C4. Some costumers are on prem, some are on Amazon Web Services (AWS). System is horizontally scalable (to a degree) and fault tolerant. Cloud patterns adoption depends on the project / client, going from rudimentary per-machine Ansible to automated blue green deployments on AWS.

C5. Multiple dozens of billions of requests processed per day.

C6. I answered single deploy, but in reality we deploy 3 times. We cluster our customer geographically and we deploy on US, Europe and Australia and each customer have their own database.

C7. Sorry for not sharing company name - you're asking for potentially sensitive information. I would not even include that question.

ID	Country	Role	Product strategy	Active monthly users	Company size
R1	Portugal	Engineer	SSMC	1k – 10k	51 – 250
R2	Portugal	CTO	SSMC	10k – 100k	11 – 50
R3	—	Engineer	SSMC	10k – 100k	1 – 10
R4	Portugal	Engineer	OSPC	< 100	> 251
R5	Spain	Team Leader	SSMC	100k – 1M	> 251
R6	Portugal	Senior Engineer	SSMC	100 – 1k	11 – 50
R7	Portugal	Engineer	SSMC	100 – 1k	> 251
R8	Portugal	Senior Engineer	SSSC	1k – 10k	> 251
R9	Portugal	Senior Engineer	SSMC	10k – 100k	11 – 50
R10	Portugal	CTO	SSMC	< 100	1 – 10
R11	Portugal	Senior Engineer	OSPC	100k – 1M	> 251
R12	U.K.	Senior Engineer	OSPC	1k – 10k	51 – 250
R13	Portugal	Engineer	SSMC	100 – 1k	1 – 10
R14	Portugal	CEO	SSMC	100 – 1k	11 – 50
R15	Canada	Developer	OSPC	10k – 100k	> 251
R16	Portugal	Engineer	SSMC	100k – 1M	51 – 250
R17	U.K.	Engineer	SSMC	> 1M	> 251
R18	Portugal	Consultant	SSMC	1k – 10k	1 – 10
R19	Portugal	Team Leader	SSMC	1k – 10k	11 – 50
R20	Portugal	CTO	OSPC	100 – 1k	11 – 50
R21	—	Engineer	SSMC	1k – 10k	1 – 10
R22	U.K.	Team Leader	SSMC	< 100	1 – 10
R23	—	Senior Engineer	OSPC	10k – 100k	11 – 50
R24	Portugal	Senior Engineer	SSMC	100 – 1k	1 – 10
R25	Portugal	Engineer	SSMC	100k – 1M	11 – 50
R26	Germany	Engineer	SSMC	100k – 1M	> 251
R27	U.S.A.	Architect	SSMC	1k – 10k	1 – 10
R28	U.K.	Senior Engineer	SSMC	> 1M	51 – 250
R29	U.K.	Senior Engineer	SSMC	100 – 1k	11 – 50
R30	Portugal	Engineer	OSPC	100k – 1M	51 – 250
R31	U.K.	Senior Engineer	SSMC	100 – 1k	11 – 50
R32	—	Product owner	SSMC	10k – 100k	> 251
R33	Portugal	Team Leader	SSMC	1k – 10k	11 – 50
R34	Portugal	Senior Engineer	SSSC	< 100	> 251
R35	Portugal	Engineer	OSPC	> 1M	1 – 10

Table B.1: Classification of the respondent from the questionnaires responses (1/3). The product strategy uses the following acronyms: One System Per Customer One system per customer (OSPC), Single System Multiple Customers Single system, multiple customers (SSMC), and Single System for Single Customer Single system, single customer (SSSC) The responses are divided in two parts. Questions not responded are identified with a "—" character.

ID	Country	Role	Product strategy	Active monthly users	Company size
R36	U.S.A.	Senior Engineer	SSMC	100k – 1M	> 251
R37	Portugal	Senior Engineer	OSPC	> 1M	51 – 250
R38	Germany	Senior Engineer	SSSC	1k – 10k	11 – 50
R39	Portugal	CEO	SSMC	< 100	1 – 10
R40	Switzerland	CTO	SSMC	100 – 1k	1 – 10
R41	Portugal	Senior Engineer	SSMC	10k – 100k	> 251
R42	Brazil	CTO	SSMC	100 – 1k	1 – 10
R43	Portugal	Senior Engineer	SSMC	100k – 1M	> 251
R44	Portugal	Engineer	SSMC	< 100	> 251
R45	—	Engineer	SSMC	100k – 1M	51 – 250
R46	U.S.A.	Senior Engineer	OSPC	100k – 1M	> 251
R47	—	Team Leader	SSSC	< 100	> 251
R48	Portugal	Senior Engineer	SSSC	1k – 10k	1 – 10
R49	U.K.	Senior Engineer	OSPC	< 100	51 – 250
R50	—	Engineer	SSSC	1k – 10k	1 – 10
R51	—	CTO	SSMC	1k – 10k	1 – 10
R52	U.K.	Engineer	SSMC	> 1M	51 – 250
R53	Portugal	Engineer	SSMC	< 100	1 – 10
R54	Portugal	Head of Innovation and Research	SSMC	100k – 1M	51 – 250
R55	Portugal	CTO	SSMC	100k – 1M	11 – 50
R56	Portugal	Senior Engineer	SSMC	< 100	1 – 10
R57	Portugal	Engineer	SSMC	< 100	11 – 50
R58	Spain	Architect	SSMC	100k – 1M	51 – 250
R59	U.S.A.	Senior Engineer	SSMC	100k – 1M	> 251
R60	—	Architect	SSSC	< 100	1 – 10
R61	U.K.	Engineer	SSMC	1k – 10k	> 251
R62	Germany	Team Leader	SSMC	10k – 100k	> 251
R63	Portugal	CTO	SSMC	< 100	1 – 10
R64	Portugal	Architect	SSSC	< 100	11 – 50
R65	U.S.A.	Engineer	OSPC	10k – 100k	11 – 50
R66	Ireland	Engineer	SSMC	> 1M	11 – 50
R67	—	Team Leader	SSMC	100 – 1k	11 – 50
R68	Netherlands	Engineer	SSMC	1k – 10k	11 – 50
R69	Portugal	Engineer	SSMC	> 1M	> 251
R70	Luxembourg	Product Manager	SSMC	100 – 1k	> 251

Table B.2: Classification of the respondent from the questionnaires responses (2/3). The product strategy uses the following acronyms: One System Per Customer OSPC, Single System Multiple Customers SSMC, and Single System for Single Customer SSSC The responses are divided in two parts. Questions not responded are identified with a "—" character.

ID	Country	Role	Product strategy	Active monthly users	Company size
R71	Italy	Senior Engineer	SSMC	10k – 100k	11 – 50
R72	U.K.	CTO	SSMC	< 100	1 – 10
R73	—	Team Leader	SSSC	100 – 1k	51 – 250
R74	—	Senior Engineer	SSMC	100 – 1k	1 – 10
R75	Netherlands	Senior Engineer	SSMC	1k – 10k	1 – 10
R76	—	Engineer	OSPC	100k – 1M	51 – 250
R77	—	Engineer	OSPC	100 – 1k	11 – 50
R78	—	Team Leader	SSMC	1k – 10k	> 251
R79	U.K.	Senior Engineer	OSPC	< 100	11 – 50
R80	U.S.A.	Architect	SSMC	100k – 1M	> 251
R81	U.K.	Senior Engineer	SSMC	10k – 100k	> 251
R82	Ireland	Engineer	SSMC	10k – 100k	> 251
R83	Portugal	Senior Engineer	SSMC	100k – 1M	11 – 50
R84	Portugal	CTO	SSSC	100k – 1M	1 – 10
R85	U.K.	Head of Reliability Engineering	SSMC	10k – 100k	> 251
R86	—	Engineer	SSMC	100k – 1M	> 251
R87	—	Architect	OSPC	1k – 10k	> 251
R88	Andorra	Engineer	OSPC	1k – 10k	1 – 10
R89	U.S.A.	Senior Engineer	SSMC	100k – 1M	51 – 250
R90	U.K.	CTO	SSMC	10k – 100k	11 – 50
R91	U.S.A.	Dev Rel	SSMC	1k – 10k	11 – 50
R92	—	Architect	SSMC	10k – 100k	51 – 250
R93	Japan	Senior Engineer	OSPC	100 – 1k	11 – 50
R94	India	Engineer	SSMC	10k – 100k	> 251
R95	Portugal	CTO	SSMC	100 – 1k	11 – 50
R96	Portugal	Team Leader	OSPC	1k – 10k	> 251
R97	Portugal	CEO	SSMC	100 – 1k	1 – 10
R98	Germany	Product Owner	OSPC	< 100	51 – 250
R99	Singapore	Engineer	SSMC	100k – 1M	11 – 50
R100	U.K.	CTO	SSMC	> 1M	1 – 10
R101	Denmark	Senior Engineer	SSMC	1k – 10k	51 – 250
R102	Portugal	Engineer	SSMC	< 100	1 – 10

Table B.3: Classification of the respondent from the questionnaires responses (3/3). The product strategy uses the following acronyms: One System Per Customer OSPC, Single System Multiple Customers SSMC, and Single System for Single Customer SSSC The responses are divided in two parts. Questions not responded are identified with a "—" character.

ID	Orchestration Manager	Automated Scalability	Infrastructure as Code	Failure Injection	Containerization	Automated Recovery	Job Scheduler	Messaging System	Service Discovery	Preemptive Logging	Logging Aggregation	External Monitor
R1	○	○	○	○	○	●	●	●	—	●	●	○
R2	○	○	○	○	○	○	—	●	○	○	○	○
R3	○	○	○	○	○	○	○	○	○	○	○	○
R4	○	○	●	○	●	○	○	○	○	—	●	○
R5	●	●	●	○	●	●	○	●	●	●	●	●
R6	○	○	○	○	●	—	○	○	○	○	●	○
R7	○	●	●	●	○	●	●	●	●	●	●	●
R8	●	●	○	○	●	●	○	○	○	●	●	●
R9	○	●	○	○	○	○	○	○	○	●	○	○
R10	●	●	●	●	●	●	●	●	●	○	○	○
R11	●	○	●	●	●	●	●	●	○	●	●	—
R12	●	●	●	○	●	●	●	●	●	○	●	●
R13	—	●	●	○	●	—	—	●	●	●	●	●
R14	●	●	●	○	●	○	●	●	○	○	●	●
R15	●	●	○	—	○	○	○	●	●	—	●	—
R16	○	●	●	○	●	○	●	●	○	○	○	●
R17	●	●	●	●	●	●	●	●	●	●	●	—
R18	○	●	●	○	●	○	○	○	○	—	●	●
R19	○	○	○	○	●	○	●	○	○	○	○	●
R20	○	○	●	○	●	○	○	○	○	○	●	●
R21	○	●	●	—	○	●	●	●	●	●	●	●
R22	●	●	●	○	○	○	●	○	○	○	●	●
R23	○	○	●	○	●	○	○	○	○	○	○	○
R24	●	●	●	○	○	○	●	●	○	○	●	●
R25	○	○	●	—	○	●	○	●	○	●	●	—
R26	○	○	●	○	○	○	●	●	○	○	●	●
R27	●	●	●	○	●	●	●	○	●	○	●	●
R28	●	●	●	○	●	●	●	●	○	○	●	●
R29	●	●	●	○	●	○	○	○	●	○	●	●
R30	—	●	—	—	●	●	●	—	—	●	●	—
R31	○	○	○	○	●	○	●	●	●	●	●	○
R32	○	●	—	—	○	●	○	—	—	—	—	—
R33	○	●	●	○	●	●	—	●	●	○	●	●
R34	○	●	●	●	●	●	●	●	●	○	●	●
R35	●	●	●	○	●	●	●	●	●	●	●	●

Table B.4: Pattern adoption per respondent (1/3). Positive response are identified with ●, ○ identified negative responses and "—" is used when the user did not provide a response to the question.

ID	Orchestration Manager	Automated Scalability	Infrastructure as Code	Failure Injection	Containerization	Automated Recovery	Job Scheduler	Messaging System	Service Discovery	Preemptive Logging	Logging Aggregation	External Monitor
R36	●	●	●	●	●	●	●	●	●	●	●	●
R37	●	●	●	○	○	●	○	●	●	●	●	○
R38	●	○	●	○	●	●	○	●	●	●	●	●
R39	○	○	○	○	○	○	○	●	○	—	○	●
R40	○	○	●	○	●	○	○	●	○	○	●	●
R41	●	○	●	○	○	○	○	●	●	●	●	●
R42	●	●	●	○	●	●	○	●	○	○	●	●
R43	●	○	●	○	●	○	○	●	●	—	●	—
R44	—	—	—	—	—	—	—	—	—	—	—	—
R45	●	—	●	●	—	●	●	●	—	●	●	—
R46	●	●	●	○	●	●	●	●	●	○	●	●
R47	○	○	○	○	○	○	○	○	○	○	○	
R48	●	○	○	○	●	○	●	○	○	●	●	●
R49	○	○	○	○	●	○	○	○	●	●	●	○
R50	○	●	●	○	○	○	●	●	●	●	●	○
R51	●	●	●	●	○	○	●	●	●	●	●	○
R52	●	●	●	—	●	—	—	—	—	●	●	—
R53	○	●	○	○	●	○	○	○	—	○	○	○
R54	○	●	○	—	○	—	●	●	○	●	●	—
R55	●	●	●	○	○	●	○	○	○	○	○	●
R56	○	●	●	○	●	○	○	●	●	○	○	○
R57	○	○	●	○	●	○	○	●	○	●	○	○
R58	●	●	●	○	●	○	○	●	●	—	●	●
R59	○	○	●	○	●	●	●	●	○	○	●	●
R60	○	●	●	○	●	○	○	●	●	○	○	○
R61	○	●	—	○	●	—	●	●	●	—	●	—
R62	●	●	●	●	●	●	●	●	●	●	●	●
R63	○	●	●	○	○	○	●	○	○	○	○	●
R64	—	●	●	○	●	○	○	●	○	○	○	○
R65	●	●	●	○	●	●	●	●	●	○	●	●
R66	—	○	—	—	○	—	●	●	○	—	—	●
R67	○	○	○	○	○	○	●	●	○	●	●	●
R68	●	●	●	○	●	—	●	○	○	●	●	○
R69	●	●	●	●	●	●	●	●	●	●	●	●
R70	○	●	○	○	●	○	○	○	○	●	—	●

Table B.5: Pattern adoption per respondent (2/3). Positive response are identified with ●, ○ identified negative responses and "—" is used when the user did not provide a response to the question.

ID	Orchestration Manager	Automated Scalability	Infrastructure as Code	Failure Injection	Containerization	Automated Recovery	Job Scheduler	Messaging System	Service Discovery	Preemptive Logging	Logging Aggregation	External Monitor
R71	○	●	○	○	●	●	●	●	○	●	●	●
R72	●	●	●	○	●	○	●	○	●	○	●	○
R73	—	—	—	—	—	●	●	—	○	—	—	●
R74	●	●	●	○	○	●	○	—	○	—	○	●
R75	●	●	○	○	●	●	●	●	●	●	●	●
R76	○	○	○	○	●	○	●	●	●	○	○	●
R77	●	●	●	○	●	●	●	●	●	—	—	●
R78	●	●	●	●	●	○	●	●	●	○	●	●
R79	●	●	●	○	●	●	●	●	●	●	●	●
R80	○	○	●	●	●	○	○	●	●	●	●	●
R81	●	●	●	○	●	●	●	●	●	●	●	○
R82	○	○	●	○	○	●	—	●	—	—	●	○
R83	—	●	●	●	●	●	●	●	●	●	●	●
R84	○	●	●	●	○	●	●	●	○	●	●	○
R85	●	●	●	○	●	●	●	●	●	●	●	●
R86	●	●	●	—	●	●	●	●	—	○	●	—
R87	●	●	●	○	●	○	●	●	○	●	●	○
R88	○	○	○	○	○	○	○	○	○	○	○	●
R89	●	○	●	●	●	●	●	●	●	●	●	●
R90	●	●	●	○	●	●	●	●	●	●	●	●
R91	●	●	○	●	●	●	○	○	○	○	●	●
R92	○	○	○	○	○	○	●	●	○	○	○	○
R93	●	●	●	●	●	●	●	●	●	●	●	●
R94	○	●	—	—	●	○	●	●	●	—	—	—
R95	●	●	●	○	●	○	●	●	○	○	●	●
R96	●	●	●	○	●	●	●	●	●	○	●	●
R97	○	●	●	—	○	—	○	●	—	—	●	●
R98	●	○	●	○	●	○	●	●	●	○	●	○
R99	○	●	●	—	●	●	●	●	○	●	●	○
R100	●	●	●	●	●	●	●	●	●	●	●	●
R101	●	●	●	○	●	●	●	●	●	○	●	●
R102	○	○	○	○	○	●	●	●	○	●	○	○

Table B.6: Pattern adoption per respondent (3/3). Positive response are identified with ●, ○ identified negative responses and "—" is used when the user did not provide a response to the question.

Appendix C

Publications

During this research, the author published 18 peer-reviewed papers and supervised 10 Bachelor and Master's students. These resulted in 119 citations, an h-index of 5, and an i10-index of 3[1]. The next sections detail the publications resulting from this research, other publications from the author, and his (co-)supervisions.

C.1 Publications Resulting from this Research

Contributions from this research led to the publication of several peer-reviewed papers, listed in Table C.1 (p. 256).

C.1.1 Patterns for Software Orchestration on the Cloud

22nd Pattern Languages of Programs. Pittsburgh, Pennsylvania, USA. 2015.

Abstract: Software businesses are redirecting their expansion towards service-oriented business models, highly supported by cloud computing. While cloud computing is not a new research subject, there is a clear lack of documented best practices on how to orchestrate cloud environments, either public, private or hybrid. This paper is targeted at DevOps practitioners and explores solutions for cloud orchestration, describing them

[1] Using Google Scholar as reference, according to https://scholar.google.pt/citations?hl=en&user=Q6Dv2ZcAAAAJ

Title	Citations	Year
A Survey on the Adoption of Patterns for Engineering Software for the Cloud	—	submitted
Design Patterns for Cloud Computing	—	submitted
Overview of a Pattern Language for Engineering Software for the Cloud	2	2018
Engineering Software for the Cloud: External Monitoring and Failure Injection	2	2018
Engineering Software for the Cloud: Automated Recovery and Scheduler	2	2018
Engineering Software for the Cloud: Messaging Systems and Logging	6	2017
Engineering Software for the Cloud: Patterns and Sequences	5	2016
Patterns for Software Orchestration on the Clouds	14	2015

Table C.1: Peer-reviewed published work from the author. The publication count was obtained from Google Scholar on December 12th, 2019.

as three patterns: a) SOFTWARE CONTAINERIZATION, providing resource sharing with minimal virtualization overhead, b) LOCAL REVERSE PROXY, allowing applications to access any service in a cluster abstracting its placement and c) ORCHESTRATION BY RESOURCE OFFERING, ensuring applications get orchestrated in a machine with the required resources to run it. The authors believe that these three DevOps patterns will help researchers and newcomers to cloud orchestration to identify and adopt existing best practices earlier, hence, simplifying software life cycle management. [BCS15]

C.1.2 Engineering Software for the Cloud: Patterns and Sequences

11th Latin American Conference on Pattern Languages of Programs (SugarLoaf PLoP). Buenos Aires, Argentina. 2016.

Abstract: Software businesses are quickly moving towards the cloud. While cloud computing is not a new research subject, engineering software for the cloud is still a challenge, demanding broad knowledge over a multitude of processes and tools that most software development teams lack. This paper identifies and briefly describes the practices required to efficiently engineer software for the cloud. The authors use the concept of patterns to capture and share those practices and describe their possible usage in an exemplar sequence. Patterns are aggregated into categories, namely: *development*, *deployment*, *execution*, *discovery and communication*, *monitoring* and *supervision*. An example sequence of application for these patterns is described. The paper is targeted at newcomers, practitioners and expert developers of software for the cloud, guiding them

through architectural decisions, at solving specific issues or just validating their decisions. [Bol+16]

C.1.3 Engineering Software for the Cloud: Messaging Systems and Logging

22nd European Conference on Pattern Languages of Programs (EuroPLoP). Irsee, Bavaria, Germany. 2017.

Abstract: Software business continues to expand globally, highly motivated by the reachability of the Internet and possibilities of Cloud Computing. While widely adopted, development for the cloud has some intrinsic properties to it, making it complex to any newcomer. This research is capturing those intricacies using a pattern catalog, with this paper contributing with three of those patterns: Messaging System, a message bus for abstracting service placement in a cluster and orchestrating messages between multiple services; Preemptive Logging, a design principle where services and servers continuously output relevant information to log files, making them available for later debugging failures; and Log Aggregation, a technique to aggregate logs from multiple services and servers in a centralized location, which indexes and provides them in a queryable, user friendly format. These patterns are useful for anyone designing software for the cloud, either to guide or validate their design decisions. [Bol+17]

C.1.4 Engineering Software for the Cloud: External Monitoring and Fault Injection

23rd European Conference on Pattern Languages of Programs (EuroPLoP). Irsee, Bavaria, Germany. 2018.

Abstract: Cloud software continues to expand globally, highly motivated by the how widespread the Internet is and the possibilities it unlocks with Cloud Computing. Still, cloud development has some intrinsic properties to it, making it complex to unexperienced developers. This research is capturing those intricacies in the form of a pattern language, gathering over 12 patterns for engineering software for the cloud. This paper elaborates on that research by contributing with two new patterns: External Monitoring, which continuously monitors the system as a black box, validating its status and Fault injection, which continuously verifies system reliability by injecting failures into the cloud environment and confirming that the system recovers from it. The described patterns are

useful for anyone designing software for the cloud, either to bootstrap or validate their design decisions and ultimately enable them to create better software. [Bol+18b]

C.1.5 Engineering Software for the Cloud: Automated Recovery and Scheduler

23rd European Conference on Pattern Languages of Programs (EuroPLoP). Irsee, Bavaria, Germany. 2018.

Abstract: Cloud software continues to expand globally, highly motivated by the how widespread the Internet is and the possibilities it unlocks with Cloud Computing. Still, cloud development has some intrinsic properties to it, making it complex to unexperienced developers. This research is capturing those intricacies in the form of a pattern language which gathers over 12 patterns for engineering software for the cloud. This paper elaborates on that research by contributing with two new patterns: Automated Recovery which checks if a container is working properly, automatically recovering it in case of failure and Scheduler, which periodically executes actions within the infrastructure. The described patterns are useful for anyone designing software for the cloud, either to bootstrap or validate their design decisions and ultimately enable them to create better software. [Bol+18a]

C.1.6 Overview of a Pattern Language for Engineering Software for the Cloud

25th Pattern Languages of Programs (PLoP). Portland, Oregon, USA. 2018.

Abstract: Software businesses are continuously increasing their cloud presence in the cloud. While cloud computing is not a new research topic, designing software for the cloud still requires engineers to make an investment to become proficient working with it. This paper introduces a pattern language for cloud software development and briefly describes details pattern. Design patterns can help developers validate or design their cloud software. The language is composed by ten patterns novel patterns organizes in three categories: Orchestration and Supervision, Monitoring and Discovery and Communication. Finally, the paper demonstrates how to adopt the pattern language using a pattern application sequence. [SFC18]

C.1.7 Design Patterns for Cloud Computing

Submitted to Springer's Lecture Notes of Computer Science journal Transactions on Pattern Languages of Programming, ISSN 1869-6015. Pending acceptance and publication.
Abstract: Software businesses are continuously increasing their presence in the cloud. While cloud computing is not a new research topic, designing software for the cloud is still a challenge, requiring from engineers a vast investment in research to become proficient at working with it. To facilitate cloud adoption, design patterns can be used, as they provide valuable design knowledge and implementation guidelines for recurrent engineering problems. This work introduces a pattern language for designing software for the cloud. We believe developers can significantly reduce their Research and Development (R&D) time by adopting these patterns to bootstrap their cloud architecture. The language is composed by 10 patterns, organized into four categories: Automated Infrastructure Managements, Orchestration and Supervision, Monitoring, and Discovery and Communication.

C.2 Other Publications from the Author

Contributions to other lines of research, along with student supervision, led to the publication of several peer reviewed papers, listed in Table C.2 (p. 260).

C.2.1 Dataflow Programming: Concept, Languages and Applications

7^{th} Doctoral Symposium in Informatics Engineering. Lisbon, Portugal, 2012.
Abstract: Dataflow Programming (DFP) has been a research topic of Software Engineering since the '70s. The paradigm models computer pro- grams as a direct graph, promoting the application of dataflow diagram principles to computation, opposing the more linear and classical Von Neumann model. DFP is the core to most visual programming languages, which claim to be able to provide end-user programming: with it's visual interface, it allows non-technical users to extend or create applications without programming knowledges. Also, DFP is capable of achieving parallelization of computation without introducing development complexity, resulting in an increased performance of applications built with it when using multi-core computers. This survey describes how visual programming languages built on top of DFP can be used for end-user programming and how easy it is to achieve concurrency by applying the paradigm, without any development overhead. DFP's open problems are discussed and some guidelines for adopting the paradigm are provided. [Sou12]

Title	Citations	Year
Towards a pattern language for the masters student	—	2019
Testing and deployment patterns for the internet-of-things	—	2019
A Testing and Certification Methodology for an Open Ambient-Assisted Living Ecosystem	7	2014
Collaborative Web Platform for UNIX-Based Big Data Processing	—	2014
Sensors, actuators and services: a distributed approach	—	2013
A testing and certification methodology for an Ambient-Assisted Living ecosystem	4	2013
Monitor, Control and Process – An Adaptive Platform for Ubiquitous Computing	—	2013
Object-Functional Patterns: Re-thinking Development in a Post-Functional World	—	2012
Ubiquitous ambient assisted living solution to promote safer independent living in older adults suffering from co-morbidity	26	2012
A Collaborative Expandable Framework for Software End-Users and Programmers	—	2012
Scalable Integration of Multiple Health Sensor Data for Observing Medical Patterns	4	2012
Dataflow Programming: Concept, Languages and Applications	47	2012

Table C.2: Peer-reviewed published work from the author not directly related to this research. The publication count was obtained from Google Scholar on December 12th, 2019.

C.2.2 Scalable Integration of Multiple Health Sensor Data for Observing Medical Patterns

9th Cooperative Design, Visualization, and Engineering Conference. Osaka, Japan. 2012.

Abstract: With an aging global population, Ambient Assisted Living (aal) attempts to improve life expectancy and quality of life through the remote monitoring of various health signals using personal and home-based sensors. Possible medical conditions can be early ascertained by observable patterns over the patients' health data. However, aggregating multiple raw signals and matching against medical protocols can be computational and bandwidth intensive. Moreover, adding new protocols requires non-trivial expertise to define necessary rules. This paper describes a lightweight, scalable, and composable mechanism that captures, processes and infers possible health problems from raw data obtained from multiple sensors. [FSM12]

C.2.3 A Collaborative Expandable Framework for Software End-Users and Programmers

9th Cooperative Design, Visualization, and Engineering Conference. Osaka, Japan. 2012.

Abstract: Monitor, control and process data on top of distributed networks has been a trending topic in the past few years, with ubiquity being adjective to computing and, gradually, the Internet of Things becoming a reality in home and factory automation or Ambient Assisted Living (aal). Still, there is a general lack of knowledge and best practices on how to build systems that integrate devices and services from third-parties which connect dynamically with each other. Recurring problems such as security, clustering, message passing, deployment and other orchestration details also lack a standardized solution. The authors describe a platform that simplifies the bootstrap and maintenance of such complex systems, presenting its application in an aal scenario. Such platform could orchestrate most distributed systems, possibly setting a pattern for distributed ubiquitous computing. [AFB12]

C.2.4 Ubiquitous ambient assisted living solution to promote safer independent living in older adults suffering from co-morbidity

34th Conference of the IEEE Engineering in Medicine and Biology Society (EMBC). San Diego, CA, USA. 2012.

Abstract: This paper describes the development, deployment and trial results from 9 volunteers using the eCAALYX system. The eCAALYX system is an ambient assisted living telemonitoring system aimed at older adults suffering with co-morbidity. Described is a raw account of the challenges that exist and results in bringing a Telemedicine system from laboratory to real-world implementation and results for usability, functionality and reliability. [Pre+12b]

C.2.5 Object-Functional Patterns: Re-thinking Development in a Post-Functional World

8th Internal Conference on Quality of Information and Communications Technology (QUATIC). Lisbon, Portugal. 2012.

Abstract: Programing paradigms define how to think and design while creating software. Object-Oriented and Functional paradigms are two of the most adopted for synthesizing it. Modern languages, attempting to provide higher abstractions, are increasingly supporting native multi-paradigm programming styles. The Object-functional approach still uses

classes for information and high-level structure, but allows algorithms to be implemented functionally. New challenges now exist and there is a general lack of knowledge on best practices for adopting this paradigm. This research proposes the systematic usage of software patterns to capture these new recurring problems and their solutions, though not discarding the identification of new algorithms and designs. We will use Scala as a base language, and will attempt to validate our hypothesis through multiple methodologies, including quasi-experiments and case studies. We expect to provide a basis for improvement for programming languages (through pattern absorption) and for software engineering professionals. [BF12]

C.2.6 Monitor, Control and Process – An Adaptive Platform for Ubiquitous Computing

10^{th} Cooperative Design, Visualization, and Engineering Conference. Mallorca, Spain. 2013.

Abstract: Monitor, control and process data on top of distributed networks has been a trending topic in the past few years, with ubiquity being adjective to computing and, gradually, the Internet of Things becoming a reality in home and factory automation or Ambient Assisted Living (aal). Still, there is a general lack of knowledge and best practices on how to build systems that integrate devices and services from third-parties which connect dynamically with each other. Recurring problems such as security, clustering, message passing, deployment and other orchestration details also lack a standardized solution. The authors describe a platform that simplifies the bootstrap and maintenance of such complex systems, presenting its application in an aal scenario. Such platform could orchestrate most distributed systems, possibly setting a pattern for distributed ubiquitous computing. [SM13]

C.2.7 Sensors, Actuators and Services: a Distributed Approach

13^{th} conference on Systems, Programming, Languages, and Applications: Software for Humanity. Indianapolis. 2013.

Abstract: Proliferation of the Internet is enabling the use of sensors and actuators to capture data and control devices remotely in a multitude of domains. Still, there is a general lack of best practices while designing such large scale real-time systems. This paper describes a generic architecture used on the implementation of a framework for deploying such systems in the cloud, enabling run-time evolution of the system with

new sensors, actuators or services possibly developed by third-parties being integrated dynamically. Such architecture orchestrates the flow of information in the ecosystem and scales transparently to external components when needed, requiring no change in them. Adoption in the Portuguese nation-wide AAL project AAL4ALL is then described. [Sou13]

C.2.8 Collaborative Web Platform for UNIX-Based Big Data Processing

11th Cooperative Design, Visualization, and Engineering Conference. Seattle, WA, USA. 2014

Abstract: UNIX-based operative systems were always empowered by scriptable shell interfaces, with a core set of powerful tools to perform manipulation over files and data streams. However those tools can be difficult to manage at the hands of a non-expert programmer. This paper proposes the creation of a Collaborative Web Platform to easily create workflows using common UNIX command line tools for processing Big Data through a collaborative web GUI. [CFS14]

C.2.9 A Testing and Certification Methodology for an Ambient-Assisted Living Ecosystem

International Journal of E-Health and Medical Communications in 2014.

Abstract: To cope with the needs raised by the demographic changes in our society, several Ambient-Assisted Living (AAL) technologies have emerged in recent years, but those 'first offers' are often monolithic, incompatible and thus expensive and potentially not sustainable. The AAL4ALL project aims at improving that situation through the development of an open ecosystem of interoperable products and services for AAL, tied together via an integration infrastructure. To that end, the project encompasses the specification of a set of reference models and requirements for interoperable products and services, against which candidate products and services can be tested and certified, and subsequently integrated as components of the ecosystem. This paper proposes a testing and certification methodology for such an ecosystem. [Far+13]

C.2.10 Testing and Deployment Patterns for the Internet-of-Things

24th European Conference on Pattern Languages of Programs. Irsee, Bavaria, Germany.

Abstract: As with every software, Internet-of-Things (IoT) systems have their own

life-cycle, from conception to construction, deployment, and operation. However, the testing requirements from these systems are slightly different due to their inherent coupling with hardware and human factors. Hence, the procedure of delivering new software versions in a continuous integration/delivery fashion must be adopted. Based on existent solutions (and inspired in other closely-related domains), we describe two common strategies that developers can use for testing IoT systems, (1) Testbed and (2) Simulation-based Testing, as well as one recurrent solution for its deployment (3) Middleman Update. [DFS19]

C.2.11 Towards a Pattern Language for Writing Engineering Theses

24th European Conference on Pattern Languages of Programs. Irsee, Bavaria, Germany.
Abstract: Every year, thousands of new students begin their Masters in Science dissertation in computer science/engineering and other Science, technology, engineering, and mathematics related topics. Despite being regarded as a common occurrence by the faculty, it represents the culmination of years of studying and preparation for their professional life. Notwithstanding, these students face well-known recurrent problems: how to choose a topic, how to choose an advisor, how to start researching, and how to deal with all the unknown associated to the contact with academic research. Although there are several books on how to write a thesis, most of them avoid prescriptive recommendations on topics beyond research *per se* or focus on doctoral students, for which the duration and motivation are significantly different. In this paper, we draft a pattern language comprised of thirty patterns that we have observed from supervising over a hundred masters students with within the last decade. [FRS19]

C.3 Supervisions

The author (co-)supervised 10 students pursuing their Bachelor and Master's degree, from Faculdade de Engenharia da Universidade do Porto (FEUP) and Instituto Superior de Engenharia do Porto (ISEP). Table C.3 (p. 265) identifies these supervisions.

Title	Work	Instituition	Author	Year
Towards DevOps: Practices and Patterns from the Portuguese Startup Scene	Master's Thesis	FEUP	Carlos Teixeira	2016
DevOps Technologies for Tomorrow	Master's Thesis	ISEP	Ruben Barros	2016
Exploring the Scala Macro System for Compile Time Model-Based Generation of Statically Type-Safe REST Services	Master's Thesis	FEUP	Filipe Oliveira	2015
Exploring Rapid Application Development for Android with Scala and SBT	Master's Thesis	FEUP	Luís Fonseca	2014
Towards a Collaborative Visual Programming Language for UNIX Workflows	Master's Thesis	FEUP	Omar Castro	2013
Towards a Self-Managed Framework for Orchestration and Integration of Devices in AAL	Master's Thesis	FEUP	João Alves	2013
Exploring the flexibility of Scala Implicits towards an Extensible Live Environment	Master's Thesis	FEUP	Vasco Grilo	2013
Aplicação Android para Monitorização de Idosos[a]	Bachelor's Project	ISEP	Vitor Moreira	2012
Aplicação Android para Monitorização de Idosos[b]	Bachelor's Project	ISEP	Diogo Silva	2012
End-User Programming In Mobile Devices through Reusable Visual Components Composition	Master's Thesis	FEUP	Tiago Almeida	2012

Table C.3: Supervised Bachelor and Master's thesis students.

[a] See Footnote *b* (p. 265).
[b] Diogo Silva and Vitor Moreira cooperated to the same project with independent contributions.

References

[18] *Key figures on Europe — Statistics Illustrated*. 2018. DOI: 10.2785/594777 (cit. on p. 211).

[Ace+13] Giuseppe Aceto et al. "Cloud monitoring: A survey." In: *Computer Networks* 57.9 (2013), pp. 2093–2115. ISSN: 13891286. DOI: 10.1016/j.comnet.2013.04.001 (cit. on pp. 29, 30, 77, 131).

[Ada15] William C. Adams. "Conducting Semi-Structured Interviews." In: *Handbook of Practical Program Evaluation: Fourth Edition* August (2015), pp. 492–505. ISSN: 0190-0447. DOI: 10.1002/9781119171386.ch19. arXiv: arXiv:1011.1669v3 (cit. on pp. 150, 151).

[AFB12] Tiago Almeida, H.S. Hugo Sereno Ferreira, and Tiago Boldt Sousa. "A Collaborative Expandable Framework for Software End-Users and Programmers." In: *9th Cooperative Design, Visualization, and Engineering*. Vol. 7467 LNCS. Osaka, Japan: Springer Berlin Heidelberg, 2012, pp. 163–166. ISBN: 9783642326080. DOI: 10.1007/978-3-642-32609-7_22 (cit. on p. 261).

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Ed. by S Ishikawa and M Silverstein. Vol. 2. Center for Environmental Structure 0. Oxford University Press, 1977. Chap. 52, pp. 503–521. ISBN: 0195019199 (cit. on pp. 5, 21).

[AJB99] Réka Albert, Hawoong Jeong, and Albert László Barabási. "Diameter of the world-wide web." In: *Nature* 401.6749 (1999), pp. 130–131. ISSN: 00280836. DOI: 10.1038/43601. arXiv: 9907038 [cond-mat]. URL: http://www.nature.com/doifinder/10.1038/43601 (cit. on p. 12).

[AKL10] Stephen Abrams, John Kunze, and David Loy. "An Emergent Micro-Services Approach to Digital Curation Infrastructure." In: *International Journal of Digital Curation* 5.1 (2010), pp. 172–186. ISSN: 1746-8256. DOI: 10.2218/ijdc.v5i1.151 (cit. on p. 17).

[Ale02] C Alexander. *The Nature of Order, Book 2: The Process of Creating Life*. Center for Environmental Structure, 2002, p. 636. ISBN: 9780972652926 (cit. on pp. 21, 74).

[Ale64] Christopher Alexander. *Notes on the Synthesis of Form*. 1964. ISBN: 0-674-62751-2 (cit. on p. 21).

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. ISBN: 0195024028 (cit. on pp. 6, 21, 71).

[AM17] Saleem Alhabash and Mengyan Ma. "A Tale of Four Platforms: Motivations and Uses of Facebook, Twitter, Instagram, and Snapchat Among College Students?" In: *Social Media and Society* 3.1 (2017). ISSN: 20563051. DOI: 10.1177/2056305117691544 (cit. on p. 194).

[Ama] Amazon. *AWS Architecture Center*. URL: https://aws.amazon.com/architecture/ (visited on 12/01/2019) (cit. on p. 36).

[Ama15] Amazon. *Amazon EC2 Container Service*. 2015. URL: https://aws.amazon.com/docker/ (visited on 12/01/2019) (cit. on p. 85).

[Ama17a] Amazon. *Amazon Cloudtrail*. 2017. URL: https://aws.amazon.com/cloudtrail/ (visited on 12/01/2019) (cit. on p. 122).

[Ama17b] Amazon. *Scheduled Tasks (cron)*. 2017. URL: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/scheduled%7B%5C_%7Dtasks.html (visited on 12/01/2019) (cit. on p. 106).

[And16] Paul Anderson. *Web 2.0 and beyond: Principles and technologies*. CRC Press, 2016, pp. 1–412. ISBN: 9781439828687. DOI: 10.5860/choice.50-3893 (cit. on pp. 2, 12).

[Arc] Arcitura Education Inc. *Dynamic Failure Detection and Recovery*. URL: http://cloudpatterns.org/design%7B%5C_%7Dpatterns/dynamic%7B%5C_%7Dfailure%7B%5C_%7Ddetection%7B%5C_%7Dand%7B%5C_%7Drecovery (visited on 12/01/2019) (cit. on p. 99).

[Arc19] Arcitura Education Inc. *Cloud Patterns*. 2019. URL: https://patterns.arcitura.com/cloud-computing-patterns (visited on 09/24/2019) (cit. on pp. 34, 122).

[Arm+10] Michael Armbrust et al. "A view of cloud computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58. ISSN: 00010782. DOI: 10.1145/1721654.1721672. URL: http://portal.acm.org/citation.cfm?doid=1721654.1721672 (cit. on pp. 30, 31).

[Atk99] Roger Atkinson. "Project management: Cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria." In: *International Journal of Project Management* 17.6 (1999), pp. 337–342. ISSN: 02637863. DOI: 10.1016/S0263-7863(98)00069-6 (cit. on p. 4).

[Azu17] Azure. *Azure Logging and Auditing*. 2017. URL: https://docs.microsoft.com/en-us/azure/security/azure-log-audit (visited on 12/01/2019) (cit. on p. 122).

[Ban+11] Prith Banerjee et al. "Everything as a service: Powering the new information economy." In: *Computer* 44.3 (2011), pp. 36–43. ISSN: 00189162. DOI: 10.1109/MC.2011.67 (cit. on pp. 2, 15).

[Bas+01] Richard Baskerville et al. "How internet software companies negotiate quality." In: *Computer* 34.5 (2001), p. 51. ISSN: 00189162. DOI: 10.1109/2.920612 (cit. on p. 18).

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. *A validation of object-oriented design metrics as quality indicators*. Tech. rep. 10. Maryland, USA: Univ. of Maryland, Dep. of Computer Science, 1996, pp. 751–761. DOI: 10.1109/32.544352 (cit. on p. 4).

[BCS15] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. "Patterns for Software Orchestration on the Cloud." In: *22nd Conference on Pattern Languages of Programs*. Pittsburgh, Pennsylvania, USA., 2015. ISBN: 9781941652039 (cit. on pp. 25, 78, 94, 143, 202, 256).

[Bec+01] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/ (cit. on p. 18).

[Bel+11] Dominique Bellenger et al. "Scaling in cloud environments." In: *Recent Researches in Computer Science - Proceedings of the 15th WSEAS International Conference on Computers, Part of the 15th WSEAS CSCC Multiconference* (2011), pp. 145–150 (cit. on p. 2).

[BF12] Tiago Boldt Sousa and Hugo Sereno Ferreira. "Object-Functional Patterns: Re-thinking Development in a Post-Functional World." In: *8th Quality of Information and Communications Technology (QUATIC)*. Lisbon, Portugal: IEEE, 2012, pp. 348–352 (cit. on p. 262).

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. 2007, p. 639. ISBN: 9780470059029. DOI: 10.1007/s13398-014-0173-7.2. arXiv: 1011.1669v3. URL: http://www.citeulike.org/group/1660/article/1686652 (cit. on p. 30).

[Bir15] Jim Bird. *Devops for Finance*. Vol. 1. 2015. ISBN: 9788578110796 (cit. on p. 30).

[BM96] Fernando Brito e Abreu and Walcelio Melo. "Evaluating the impact of object-oriented design on software quality." In: *International Software Metrics Symposium, Proceedings* (1996), pp. 90–99. DOI: 10.1109/metric.1996.492446 (cit. on p. 4).

[BMR96] F Bushmann, R Meunier, and H Rohnert. *Pattern-oriented software architecture: A System of Patterns, Volume 1*. Vol. 1. Wiley Publishing, 1996, p. 476. ISBN: 9780471958697 (cit. on p. 143).

[Bol+16] Tiago Boldt Sousa et al. "Engineering Software for the Cloud - Patterns and Sequences." In: *11th Latin American Conference on Pattern Languages of Programs Programs*. 11. Buenos Aires, Argentina, 2016, p. 8. ISBN: 9781941652053 (cit. on p. 257).

[Bol+17] Tiago Boldt Sousa et al. "Engineering Software for the Cloud: Messaging Systems and Logging." In: *22nd European Conference on Pattern Languages of Programs*. Irsee, Bavaria, Germany, 2017. ISBN: 978-1-4503-4848-5. DOI: 10.1145/3147704.3147720. URL: http://doi.acm.org/10.1145/3147704.3147720 (cit. on p. 257).

[Bol+18a] Tiago Boldt Sousa et al. "Engineering Software for the Cloud: Automated Recovery and Scheduler." In: *23rd European Conference on Pattern Languages of Programs*. Irsee, Bavaria, Germany., 2018 (cit. on p. 258).

[Bol+18b] Tiago Boldt Sousa et al. "Engineering Software for the Cloud: External Monitoring and Fault Injection." In: *23rd European Conference on Pattern Languages of Programs*. Irsee, Bavaria, Germany., 2018 (cit. on p. 258).

[Bon+14] Jonas Bonér et al. "The Reactive Manifesto (Version 2.0)." In: *Reactivemanifesto.Org* 2.16 September 2014 (2014), pp.1–2. URL: http://www.reactivemanifesto.org (cit. on pp. 106, 109).

[Bon16] Jonas Bonér. *Reactive Microservices Architecture Design Principles for Distributed Systems*. 2016, p. 54. ISBN: 978-1-491-95779-0 (cit. on pp. 29, 152).

[Boo04] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 9780201895513 (cit. on pp. 5, 21).

[BP19] Morgan Bruce and Paulo A. Pereira. *Microservices in action*. Manning Publications, 2019, p. 366. ISBN: 1617294454 (cit. on pp. 30, 170).

[Bro+14] Antonio Brogi et al. "SeaClouds." In: *ACM SIGSOFT Software Engineering Notes* 39.1 (2014), pp. 1–4. ISSN: 01635948. DOI: 10.1145/2557833.2557844 (cit. on p. 17).

[Bro15] Malcolm Bronte-stewart. "Beyond the Iron Triangle: Evaluating Aspects of Success and Failure using a Project Status Model." In: *Computing & Information Systems* 19.2 (2015), pp. 21–37 (cit. on p. 4).

[Bui15] Thanh Bui. "Analysis of Docker Security." In: *Computing Research Repository* abs/1501.0 (2015), p. 7. arXiv: 1501.02967. URL: http://arxiv.org/abs/1501.02967 (cit. on p. 88).

[Cas+11] James Casey et al. "A messaging infrastructure for WLCG." In: *Journal of Physics: Conference Series* 331.PART 6 (2011). ISSN: 17426596. DOI: 10.1088/1742-6596/331/6/062015 (cit. on pp. 30, 143).

[CC06] D Cohen and B Crabtree. *Semi-structured Interviews*. 2006. URL: http://www.qualres.org/HomeSemi-3629.html (visited on 12/01/2019) (cit. on p. 151).

[CFS14] Omar Castro, Hugo Sereno Ferreira, and Tiago Boldt Sousa. *Collaborative Web Platform for UNIX-Based Big Data Processing*. Seattle, WA, USA, 2014. DOI: 10.1007/978-3-319-10831-5_30 (cit. on p. 263).

[Cha05] Robert N. Charette. *Why Software Fails*. 2005. DOI: 10.1109/MSPEC.2005.1502528. URL: http://spectrum.ieee.org/computing/software/why-software-fails (cit. on p. 108).

[Cha17] Chaos Community. *Principles of Chaos Engineering*. 2017. URL: http://principlesofchaos.org/ (visited on 12/01/2019) (cit. on pp. 114, 115).

[Chr17] Chronos. *Chronos*. 2017. URL: https://mesos.github.io/chronos/ (visited on 12/01/2019) (cit. on p. 105).

[CLW14] John Chinneck, Marin Litoiu, and Murray Woodside. "Real-time multi-cloud management needs application awareness." In: *ICPE 2014 - Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (2014), pp. 293–296. DOI: 10.1145/2568088.2576763 (cit. on p. 17).

[CMV15] Mario Callegaro, Katja Lozar Manfreda, and Vasja Vehovar. *Web survey methodology*. Sage, 2015 (cit. on p. 192).

[Com15a] CoreOS Community. *Etcd Project Page*. 2015. URL: https://github.com/coreos/etcd (visited on 12/01/2019) (cit. on p. 147).

[Com15b] Vulcanproxy Community. *Vulcanproxy Project Page*. 2015. URL: http://www.vulcanproxy.com/ (visited on 12/01/2019) (cit. on p. 147).

[Con] Mel Conway. *Conway's Law*. URL: http://www.melconway.com/Home/Conways%7B%5C_%7DLaw.html (visited on 12/01/2019) (cit. on p. 18).

[Cor+18] Fernando Dias Correia et al. "Home-based Rehabilitation With A Novel Digital Biofeedback System versus Conventional In-person Rehabilitation after Total Knee Replacement: a feasibility study." In: *Scientific Reports* 8.1 (2018), pp. 1–12. ISSN: 20452322. DOI: 10.1038/s41598-018-29668-0 (cit. on p. 173).

[Cor15] CoreOS Community. *CoreOS Project Page*. 2015. URL: https://coreos.com/ (visited on 12/01/2019) (cit. on p. 94).

[Cou00] Mick P. Couper. "Web Surveys." In: *Public Opinion Quarterly* 64.4 (2000), pp. 464–494. ISSN: 0033362X. DOI: 10.1086/318641 (cit. on p. 214).

[Cun14] Ward Cunningham. *Let It Crash*. 2014. URL: http://wiki.c2.com/?LetItCrash (cit. on p. 109).

[Cus10] Michael Cusumano. "Cloud computing and SaaS as new computing platforms." In: *Communications of the ACM* 53.4 (2010), pp. 27–29. ISSN: 00010782. DOI: 10.1145/1721654.1721667 (cit. on pp. 16, 17).

[Cyc15] Cycligent. *Continuous Delivery Patterns for Design and Deployment*. Tech. rep. 2015 (cit. on pp. 41, 89).

[DAC15] Maximilien De Bayser, Leonardo G. Azevedo, and Renato Cerqueira. "ResearchOps: The case for DevOps in scientific applications." In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015* (2015), pp. 1398–1404. DOI: 10.1109/INM.2015.7140503 (cit. on p. 88).

[Dad18] Armon Dadgar. *What is infrastructure as code and why is it important?* 2018. URL: https://www.hashicorp.com/resources/what-is-infrastructure-as-code (visited on 12/01/2019) (cit. on p. 72).

[Dat18] DataDog. *Docker Adoption*. 2018. URL: https://www.datadoghq.com/docker-adoption/ (visited on 12/01/2019) (cit. on p. 85).

[Deb08] Patrick Debois. *Agile Infrastructure & Operations*. Toronto, 2008. URL: http://www.jedi.be/blog/2008/10/09/agile-2008-toronto-agile-infrastructure-and-operations-presentation/ (cit. on pp. 12, 19).

[DFS19] João Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. "Testing and Deployment Patterns for the Internet-of-Things." In: *24th European Conference on Pattern Languages of Programs*. Irsee, Bavaria, Germany, 2019. ISBN: 9781450362061 (cit. on p. 264).

[Día+18] Jessica Díaz et al. *DevOps in practice*. 2018, pp. 1–3. DOI: 10.1145/3234152.3234199 (cit. on pp. 19, 31).

[DiN99] Darcy DiNucci. "Fragemented Future." In: *Print* 53.4 (1999), p. 2 (cit. on p. 12).

[DJG18] Dr. Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: State of DevOps 2018 Strategies for a New Economy*. Tech. rep. 2018, p. 78. URL: https://cloudplatformonline.com/rs/248-TPC-286/images/DORA-State%20of%20DevOps.pdf (cit. on pp. 18, 26, 78).

[DL12] Elias Adriano Nogueira Da Silva and Daniel Lucrédio. "Software engineering for the cloud: A research roadmap." In: *Proceedings - 2012 Brazilian Symposium on Software Engineering, SBES 2012* September 2012 (2012), pp. 71–80. ISSN: 00010782. DOI: 10.1109/SBES.2012.12 (cit. on p. 29).

[Doc18] Docker. *Dockerfile reference*. 2018. URL: https://docs.docker.com/engine/reference/builder (cit. on p. 100).

[Duv10] Paul Duvall. "Continuous Delivery Refcardz." In: (2010), p. 497 (cit. on p. 42).

[EAD14] Floris Erich, Chintan Amrit, and Maya Daneva. *DevOps Literature Review*. Tech. rep. February. 2014, p. 27. DOI: 10.13140/2.1.5125.1201 (cit. on pp. 19, 31).

[ECN15] Thomas Erl, Robert Cope, and Amin Naserpour. *Cloud Computing Design Patterns*. 2015, p. 552. ISBN: 9780133858624 (cit. on pp. 30, 34, 72, 133).

[Ede] Jason Edelman. *Network Automation with Ansible*. ISBN: 9781491937839 (cit. on p. 31).

[Ela17] Elastic. *The Open Source Elastic Stack*. 2017. URL: https://www.elastic.co/products (visited on 12/01/2019) (cit. on p. 127).

[EMP19] Thomas Erl, Zaigham Mahmood, and Ricardo Puttini. *Cloud Computing: Concepts, Technology & Architecture*. Vol. 51. 05. 2019, pp. 51–2714–51–2714. ISBN: 9780133387520. DOI: 10.5860/choice.51-2714 (cit. on p. 34).

[Eur15] European Commission. *User guide to the SME Definition*. 2015, pp. 1–60. DOI: 10.2873/782201 (cit. on pp. 194, 198, 210).

[Far+13] Joao Pascoal Faria et al. "A testing and certification methodology for an Ambient-Assisted Living ecosystem." In: *2013 IEEE 15th International Conference on e-Health Networking, Applications and Services, Healthcom 2013*. Vol. 5. Healthcom. 2013, pp. 585–589. ISBN: 9781467358019. DOI: 10.1109/HealthCom.2013.6720744 (cit. on pp. 56, 68, 263).

[Far+14] João Pascoal Faria et al. "A testing and certification methodology for an open Ambient-Assisted Living ecosystem." In: *International Journal of E-Health and Medical Communications* 5.4 (2014), pp. 90–107. ISSN: 19473168. DOI: 10.4018/ijehmc.2014100106 (cit. on pp. 56–60, 68).

[Feh+14] Christoph Fehling et al. *Cloud Computing Patterns*. 2014, pp. 239–286. ISBN: 978-3-7091-1567-1. DOI: 10.1007/978-3-7091-1568-8 (cit. on pp. 14, 34).

[Fel+12] Wes Felter et al. *IBM Research Report An Updated Performance Comparison of VirtualMachines and Linux Containers*. Tech. rep. 2012, pp. 25482–1407. URL: http://domino.watson.ibm.com/library/CyberDig.nsf/home. (cit. on pp. 30, 85, 87).

[Fer+13] Joel L. Fernandes et al. "Performance evaluation of RESTful web services and AMQP protocol." In: *International Conference on Ubiquitous and Future Networks, ICUFN* (2013), pp. 810–815. ISSN: 21658528. DOI: 10.1109/ICUFN.2013.6614932 (cit. on p. 13).

[Fer13] Eduardo B Fernandez. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. 2013, p. 584. ISBN: 1119970482 (cit. on pp. 122, 126).

[Fou15] Apache Foundation. *Mesos Project Page*. 2015. URL: http://mesos.apache.org/ (visited on 12/01/2019) (cit. on p. 148).

[Fou19] Cloud Native Computing Foundation. *CNCF Cloud Native Interactive Landscape*. 2019. URL: https://landscape.cncf.io (visited on 12/01/2019) (cit. on p. 43).

[Fow06] Martin Fowler. *Writing Software Patterns*. 2006. URL: http://www.martinfowler.com/articles/writingPatterns.html (visited on 12/01/2019) (cit. on p. 6).

[Fow15] Martin Fowler. *Monolith First*. 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html (visited on 01/10/2020) (cit. on pp. 29, 152).

[Fow17] Martin Fowler. *What do you mean by "Event-Driven"?* 2017. URL: https://martinfowler.com/articles/201701-event-driven.html (visited on 12/01/2019) (cit. on p. 106).

[Fri14] Uwe Friedrichsen. *Patterns of Resilience*. 2014. URL: https://www.slideshare.net/ufried/patterns-of-resilience (cit. on p. 42).

[FRS19] Hugo Sereno Ferreira, André Restivo, and Tiago Boldt Sousa. "Towards a Pattern Language for the Masters Student." In: *24th European Conference on Pattern Languages of Programs*. Irsee, Bavaria, Germany, 2019. ISBN: 9781450362061. DOI: 10.1145/3361149.3361184 (cit. on pp. 229, 264).

[FSM12] Hugo Sereno Ferreira, Tiago Boldt Sousa, and Angelo Martins. "Scalable integration of multiple health sensor data for observing medical patterns." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7467 LNCS. Osaka, Japan, 2012, pp. 78–84. ISBN: 9783642326080. DOI: 10.1007/978-3-642-32609-7_11 (cit. on pp. 68, 260).

[Fu+14] Qiang Fu et al. "Where do developers log? An empirical study on logging practices in industry." In: *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings* (2014), pp. 24–33. ISSN: 02705257. DOI: 10.1145/2591062.2591175 (cit. on p. 122).

[FY99] Brian Foote and Joseph Yoder. *Big Ball of Mud*. Tech. rep. Department of Computer Science University of Illinois at Urbana-Champaign, 1999. URL: http://www.laputan.org/mud/ (cit. on p. 4).

[Gad14] Ofer Gadish. *Top 9 Reasons for Cloud Application Failure*. 2014. URL: https://webcache.googleusercontent.com/search?q=cache:wD3pBZHZnxUJ:https://www.cloudendure.com/blog/top-9-reasons-cloud-application-failure/+%7B%5C&%7Dcd=1%7B%5C&%7Dhl=en%7B%5C&%7Dct=clnk%7B%5C&%7Dgl=pt (visited on 07/01/2017) (cit. on pp. 31, 47, 78).

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Ed. by Addison-Wesley Pub Co. Addison Wesley Professional Computing Series. Addison Wesley, 1994, 395So. ISBN: 0201633612. DOI: 10.1016/j.artmed.2009.05.004 (cit. on pp. iv, vi, 6, 22).

[Gar06] Simson L Garfinkel. *An Evaluation of Amazon's Grid Computing Services : EC2, S3 and SQS*. Tech. rep. 2006, pp. 1–15 (cit. on pp. 14, 30).

[Gar12] Gartner. "Gartner Says Worldwide Software-as-a-Service Revenue to Reach $14.5 Billion in 2012." In: *Press Release* (2012). URL: http://www.gartner.com/newsroom/id/1963815 (cit. on p. 15).

[Gar14] Gartner. *Software as a Service (SaaS)*. 2014. URL: https://www.gartner.com/it-glossary/software-as-a-service-saas/ (cit. on pp. 15–17).

[Gar19] Gartner. *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019*. Tech. rep. 2019. URL: https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g (cit. on pp. 14, 15).

[Gaw02] Dieter Gawlick. "Message Queuing for Business Integration." In: *{eAI} Journal* October 2002 (2002), pp. 30–33 (cit. on p. 135).

[GB14] Nikolay Grozev and Rajkumar Buyya. "Inter-Cloud architectures and application brokering: Taxonomy and survey." In: *Software - Practice and Experience* 44.3 (2014), pp. 369–390. ISSN: 00380644. DOI: 10.1002/spe.2168 (cit. on p. 17).

[Git17] Gitlab. *Postmortem of database outage of January 31*. 2017. URL: https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/ (cit. on pp. 33, 109).

[Goa16] Sébastien Goasguen. *Docker in the Cloud*. Second Edi. O'Reilly Media, 2016, p. 41. ISBN: 9781491940976 (cit. on p. 94).

[Goo15] Google. *Google Cloud Container Service*. 2015. URL: https://cloud.google.com/container-engine/ (visited on 12/01/2019) (cit. on p. 85).

[Goo18] Google. *Reliable Task Scheduling on Google Compute Engine*. 2018. URL: https://cloud.google.com/solutions/reliable-task-scheduling-compute-engine (visited on 12/01/2019) (cit. on p. 105).

[Goo19] Google. *Google Cloud locations*. 2019. URL: https://cloud.google.com/about/locations/ (cit. on p. 17).

[Gra11] Cd Graziano. "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project." In: *Master's Thesis* (2011) (cit. on p. 77).

[Gre+08] Albert Greenberg et al. "The cost of a cloud." In: *ACM SIGCOMM Computer Communication Review* 39.1 (2008), p. 68. ISSN: 01464833. DOI: 10.1145/1496091.1496103 (cit. on pp. 12, 30, 31).

[Gro16] The Open Group. *Cloud Computing Governance Framework*. 2016. URL: https://www.opengroup.org/company-reviews (visited on 12/01/2019) (cit. on p. 27).

[GŠH16] Krešimir Grgić, Ivan Špeh, and Ivan Hedi. "A web-based IoT solution for monitoring data using MQTT protocol." In: *Proceedings of 2016 International Conference on Smart Systems and Technologies, SST 2016*. IEEE Computer Society, 2016, pp. 249–253. ISBN: 9781509037186. DOI: 10.1109/SST.2016.7765668 (cit. on p. 143).

[Guc17] Sam Guckenheimer. *What is Infrastructure as Code?* 2017. URL: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code (visited on 12/01/2019) (cit. on p. 72).

[Han98] Robert Hanmer. "An Input and Output Pattern Language." In: *Design Patterns in Communications Software*. c. Cambridge University Press, 1998, pp. 95–129. ISBN: 0-521-79040-9 (cit. on pp. 23, 143).

[HB09] Margaret Harrell and Melissa Bradley. *Data Collection Methods Semi-Structured Interviews and Focus Groups*. 2009, p. 148. ISBN: 9780833048899 (cit. on p. 151).

[Her+16] Peter Herrmann et al. "Collaborative model-based development of a remote train monitoring system." In: *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*. 2016, pp. 383–390. ISBN: 9789897581892. DOI: 10.5220/0005929403830390 (cit. on p. 143).

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. "Elasticity in Cloud Computing : What It Is , and What It Is Not." In: *Presented as part of the 10th International Conference on Autonomic Computing* (2013), pp. 23–27. ISSN: 1540-9740. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/HeKoRe2013-ICAC-Elasticity.pdf (cit. on p. 5).

[HKZ11] Benjamin Hindman, Andy Konwinski, and Matei Zaharia. "Mesos: A platform for fine-grained resource sharing in the data center." In: *Proceedings of the ...* (2011), p. 32. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488 (cit. on p. 94).

[HM] Edward Hieatt and Rob Mee. *Repository Pattern*. URL: https://martinfowler.com/eaaCatalog/repository.html (visited on 12/01/2019) (cit. on p. 126).

[Hof16] Robert Hof. *Meet Project Storm, Facebook's SWAT team for disaster-proofing data centers*. 2016. URL: https://siliconangle.com/2016/08/31/meet-project-storm-facebooks-swat-team-for-disaster-proofing-data-centers/ (visited on 12/01/2019) (cit. on p. 114).

[Hol05] Erik Hollnagel. "Human reliability assessment in context." In: *Nuclear Engineering and Technology* 37 (2005), pp. 159–166 (cit. on p. 31).

[Hub+13] Markus Huber et al. "AppInspect: Large-scale evaluation of social networking apps." In: *COSN 2013 - Proceedings of the 2013 Conference on Online Social Networks* (2013), pp. 143–154. DOI: 10.1145/2512938.2512942 (cit. on p. 194).

[HW03] Gregor Hohpe and Bobby Woolf. "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions." In: *Enterprise integration patterns designing building and deploying messaging solution* (2003), p. 736. ISSN: 14602105. DOI: 10.1525/vs.2009.4.3.toc (cit. on p. 142).

[Hwa+13] Jinho Hwang et al. "A component-based performance comparison of four hypervisors." In: *Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management, IM 2013* (2013), pp. 269–276 (cit. on p. 30).

[II16] Takashi Iba and Taichi Isaku. "A Pattern Language for Creating Pattern Languages." In: *23rd Conference on Pattern Languages of Programs*. 2016 (cit. on p. 71).

[Inc15a] Mesosphere Inc. *Marathon Project Page*. 2015. URL: https://mesosphere.github.io/marathon (visited on 12/01/2019) (cit. on p. 148).

[Inc15b] Mesosphere Inc. *Mesosphere Service Discovery & Load Balancing*. 2015. URL: https://mesosphere.github.io/marathon/docs/service-discovery-load-balancing.html (visited on 12/01/2019) (cit. on p. 148).

[Ini15] Open Container Initiative. *Open Containers Project Page*. 2015. URL: http://www.opencontainers.org/ (visited on 12/01/2019) (cit. on p. 85).

[Int] International Organization for Standardization. *ISO/IEC 27000 family*. URL: https://www.iso.org/isoiec-27001-information-security.html (visited on 12/01/2019) (cit. on pp. 32, 159).

[Int19] Internetlivestats.com. *Number of Internet users in the world*. 2019. URL: http://www.internetlivestats.com/internet-users/ (cit. on pp. 2, 12).

[IO16] IEEE and The Group Open. *crontab*. 2016. URL: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html (cit. on p. 106).

[Irw67] Manley R. Irwin. "The Computer Utility: Competition or Regulation?" In: *The Yale Law Journal* 76.7 (1967), p. 1299. ISSN: 00440094. DOI: 10.2307/794825 (cit. on p. 14).

[ISM11] Takashi Iba, Mami Sakamoto, and Toko Miyake. "How to Write Tacit Knowledge as a Pattern Language: Media Design for Spontaneous and Collaborative Communities." In: *Procedia - Social and Behavioral Sciences* 26 (2011), pp. 46–54. ISSN: 18770428. DOI: 10.1016/j.sbspro.2011.10.561 (cit. on p. 23).

[Jac08] Jack Schofield. *Google angles for business users with platform as a service*. 2008. URL: https://www.theguardian.com/technology/2008/apr/17/google.software (cit. on p. 16).

[Kaw05] Barbara B. Kawulich. "Participant observation as a data collection method." In: *Forum Qualitative Sozialforschung*. Vol. 6. 2. 2005. DOI: 10.17169/fqs-6.2.466 (cit. on p. 65).

[Kim+16] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. 2016, p. 480. ISBN: 9781942788003. DOI: 2016951904 (cit. on p. 26).

[Kou18] Petros Koutoupis. *Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers*. 2018. URL: https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lxc (cit. on p. 81).

[KP10] Christian Kohls and Stefanie Panke. "Is that true...?- Thoughts on the epistemology of patterns." In: *ACM International Conference Proceeding Series* (2010), 9:1–9:14. DOI: 10.1145/1943226.1943237 (cit. on pp. 22, 23).

[Kub] Kubernetes. *Run a Stateless Application Using a Deployment*. URL: https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/ (cit. on p. 91).

[Kub17] Kubernetes. *Kubernetes Cron Jobs*. 2017. URL: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (visited on 12/01/2019) (cit. on p. 105).

[Kub18a] Kubernetes. *DNS for Services and Pods*. 2018. URL: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (cit. on p. 148).

[Kub18b] Kubernetes. *Pod Lifecycle*. 2018. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (cit. on pp. 95, 99, 100).

[Kuh99] Thomas Kuhne. "A Functional Pattern System for Object-Oriented Design." PhD thesis. 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1134%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf (cit. on p. 5).

[LA94] Barriball K. Louise and While Alison. "Collecting data using a semi-structured interview: a discussion paper." In: *Journal of Advanced Nursing* 19.2 (1994), pp. 328–335 (cit. on pp. 151, 188).

[Laz54] P.F. Lazarsfeld. "The art of asking why three principles underlying the formation of questionnaires." In: *Public opinion and propaganda: A book of readings* 1.1 (1954), pp. 26–38. DOI: 10.2307/4291274 (cit. on p. 151).

[Lew12] James Lewis. "Micro services - Java, the Unix Way." In: *33rd Degree Conference* (2012). URL: http://2012.33degree.org/pdf/JamesLewisMicroServices.pdf (cit. on p. 13).

[LF14] James Lewis and Martin Fowler. *Microservices*. 2014. URL: http://martinfowler.com/articles/microservices.html (cit. on p. 72).

[LMR01] Nelson G M Leme, Eliane Martins, and Cecília Rubira. "A Software Fault Injection Pattern System." In: *Pattern Languages of Programs*. 2001. URL: https://hillside.net/plop/plop2001/accepted%7B%5C_%7Dsubmissions/PLoP2001/ngmleme3/PLoP2001%7B%5C_%7Dngmleme3%7B%5C_%7D3.pdf (cit. on p. 114).

[Lor19] Mario Loriedo. *Containers Patterns*. 2019. URL: https://l0rd.github.io/containerspatterns/%7B%5C#%7D6 (cit. on p. 43).

[Lou12] Mike Loukides. *What is DevOps? - Infrastructure as Code*. O'Reilly Media, 2012, p. 20. ISBN: 978-1-4493-3910-4 (cit. on p. 19).

[Mag15] L. Magnoni. "Modern messaging for distributed sytems." In: *Journal of Physics: Conference Series* 608.1 (2015), p. 012038. ISSN: 17426596. DOI: 10.1088/1742-6596/608/1/012038 (cit. on p. 140).

[Mal+02] Yashwant K. Malaiya et al. "Software reliability growth with test coverage." In: *IEEE Transactions on Reliability* 51.4 (2002), pp. 420–426. ISSN: 00189529. DOI: 10.1109/TR.2002.804489 (cit. on p. 127).

[MBV06] Katja Lozar Manfreda, Zenel Batagelj, and Vasja Vehovar. "Design of Web Survey Questionnaires: Three Basic Experiments." In: *Journal of Computer-Mediated Communication* 7.3 (2006). ISSN: 1083-6101. DOI: 10.1111/j.1083-6101.2002.tb00149.x (cit. on p. 213).

[MD98a] V Marvin and R Dolores. "Experimental Models for Validating Technology." In: *Computer* May (1998), pp. 23–31 (cit. on p. 50).

[MD98b] Gerard Meszaros and Jim Doble. *A Pattern Language for Pattern Writing*. 1998. URL: http://hillside.net/index.php/a-pattern-language-for-pattern-writing (visited on 12/01/2019) (cit. on p. 69).

[Men04] Paul Menage. *CGROUPS*. Tech. rep. 2004. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (cit. on p. 87).

[Mer] Merriam-Webster English Dictionary. *Design*. URL: https://www.merriam-webster.com/dictionary/design (visited on 12/01/2019) (cit. on pp. 4, 21).

[Mes17] Mesosphere. *Marathon Health Checks*. 2017. URL: https://mesosphere.github.io/marathon/docs/health-checks.html (visited on 12/01/2019) (cit. on pp. 95, 99, 100).

[Mes18] Mesosphere. *Marathon API*. 2018. URL: https://docs.mesosphere.com/1.11/deploying-services/marathon-api/ (cit. on p. 91).

[MG11] Peter Mell and Timothy Grance. "The NIST definition of cloud computing." In: *NIST Special Publication* (2011) (cit. on pp. 15, 16).

[Mic17a] Microsoft. *Health Endpoint Monitoring pattern*. 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring (visited on 12/01/2019) (cit. on p. 133).

[Mic17b] Microsoft. *Microsoft Azure Scheduler*. 2017. URL: https://azure.microsoft.com/en-us/services/scheduler/ (visited on 12/01/2019) (cit. on pp. 105, 106).

[Mic19] Microsoft. *Cloud Design Patterns*. 2019. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/ (visited on 12/01/2019) (cit. on pp. 29, 30, 36, 38, 39).

[MK10] Shivaji P. Mirashe and N. V. Kalyankar. "Cloud Computing." In: 2.3 (2010), pp. 78–82. arXiv: 1003.4074v1 (cit. on pp. 14, 17, 27).

[Mor15] Kief Morris. *Infrastructure as Code*. O'Reilly Media, Inc., 2015. ISBN: 9781491924334 (cit. on p. 72).

[Mou15] Adrian Mouat. *Docker Security*. Tech. rep. 2015. URL: http://www.oreilly.com/webops-perf/free/docker-security.csp (cit. on p. 87).

[MR06] Catherine Marshall and Gretchen B. Rossman. "Designing qualitative research. Sage Publication." In: *Newbury Park, California* (2006), p. 114 (cit. on p. 65).

[MT10] Nenad Medvidovic and Richard N. Taylor. "Software architecture: Foundations, theory, and practice." In: *Proceedings - International Conference on Software Engineering* 2 (2010), pp. 471–472. ISSN: 02705257. DOI: 10.1145/1810295.1810435 (cit. on p. 4).

[Mur07] San Murugesan. "Understanding Web 2.0." In: *IT Professional* 9.4 (2007), pp. 34–41. ISSN: 15209202. DOI: 10.1109/MITP.2007.78 (cit. on p. 12).

[Nat03] Yefim V Natis. "Service-Oriented Architecture Scenario." In: *Gartner Research* AV-19-6751.April (2003), p. 6 (cit. on p. 13).

[Net11] Netflix. *The Netflix Simian Army*. 2011. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116 (cit. on p. 114).

[Net17] Netflix. *Chaos Monkey*. 2017. URL: https://github.com/Netflix/chaosmonkey (visited on 12/01/2019) (cit. on p. 114).

[NS14] Dmitry Namiot and Manfred Sneps-Sneppe. "On Micro-services Architecture." In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27. ISSN: 2307-8162 (cit. on pp. 12, 72).

[Ode14] Andrew Odewahn. *Field Guide To The Distributed Development Stack*. Vol. 53. 2014, p. 160. ISBN: 9788578110796. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3 (cit. on p. 16).

[Owe02] L Owens. *Introduction To Survey Research Design*. 2002. URL: http://www.researchgate.net/profile/Linda%7B%5C_%7DOwens/publication/253282490%7B%5C_%7DINTRODUCTION%7B%5C_%7DTO%7B%5C_%7DSURVEY%7B%5C_%7DRESEARCH%7B%5C_%7DDESIGN/links/545a5e1d0cf2c46f6642734c.pdf (cit. on p. 192).

[Pau+14] Subharthi Paul et al. "Application delivery in multi-cloud environments using software defined networking." In: *Computer Networks* 68 (2014), pp. 166–186. ISSN: 13891286. DOI: 10.1016/j.comnet.2013.12.005 (cit. on p. 17).

[Pin17] Pingdom. *Pingdom*. 2017. URL: https://www.pingdom.com/ (visited on 12/01/2019) (cit. on p. 133).

[Piv07] Pivotal. *RabbitMQ Tutorials*. 2007. URL: https://rabbitmq.docs.pivotal.io/35/rabbit-web-docs/tutorials/tutorial-one-java.html (visited on 12/01/2019) (cit. on p. 139).

[Pos16] Christian Posta. *Microservices for Java Developers. A Hands-on Introduction to Frameworks and Containers*. 2016, p. 129. ISBN: 9781491962077 (cit. on p. 18).

[Pre+12a] Sandra Prescher et al. "Ubiquitous ambient assisted living solution to promote safer independent living in older adults suffering from co-morbidity." In: *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*. SanDiego, CA, USA: IEEE, 2012, pp. 5118–5121. ISBN: 9781424441198. DOI: 10.1109/EMBC.2012.6347145 (cit. on p. 68).

[Pre+12b] Sandra Prescher et al. "Ubiquitous ambient assisted living solution to promote safer independent living in older adults suffering from co-morbidity." In: *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS* 2012 (2012), pp. 5118–5121. ISSN: 1557170X. DOI: 10.1109/EMBC.2012.6347145 (cit. on pp. 68, 261).

[Pun+03] Teade Punter et al. "Conducting on-line surveys in software engineering." In: *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* IEEE. 2003, pp. 80–88 (cit. on p. 192).

[Put+15] Deepak Puthal et al. "Cloud computing features, issues, and challenges: A big picture." In: *Proceedings - 1st International Conference on Computational Intelligence and Networks, CINE 2015* (2015), pp. 116–123. ISSN: 2375-5822. DOI: 10.1109/CINE.2015.31 (cit. on pp. 26, 29–31).

[PWB07] Eduardo Pinheiro, WD Weber, and LA Barroso. "Failure trends in a large disk drive population." In: *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*. Vol. 7. February. 2007, pp. 17–29 (cit. on pp. 78, 108).

[Rab20] RabbitMQ. *Queues*. 2020. URL: https://www.rabbitmq.com/queues.html (visited on 01/01/2020) (cit. on p. 140).

[RBK13] C Roderick, L Burdzanowski, and G Kruk. *The CERN Accelerator Logging Service- 10 Years in Operation: A Look at the Past, Present and Future*. Tech. rep. CERN, 2013. URL: http://cds.cern.ch/record/1611082 (cit. on p. 127).

[RD10] Nathan Regola and Jean Christophe Ducom. "Recommendations for virtualization technologies in high performance computing." In: *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010* (2010), pp. 409–416. DOI: 10.1109/CloudCom.2010.71 (cit. on p. 87).

[Rel17] New Relic. *New Relic*. 2017. URL: https://newrelic.com/ (cit. on p. 133).

[RES10] S. Ramgovind, M. M. Eloff, and E. Smith. "The management of security in cloud computing." In: *Proceedings of the 2010 Information Security for South Africa Conference, ISSA 2010* (2010), pp. 1–7. DOI: 10.1109/ISSA.2010.5588290 (cit. on pp. 27, 30).

[Ric] Mark Richards. *Microservices AntiPatterns and Pitfalls*. ISBN: 9781491963319 (cit. on p. 142).

[Ric15] Mark Richards. *Microservices versus SOA*. 2015, pp. 1–55. ISBN: 9781491941614. URL: https://www.nginx.com/microservices-soa/ (cit. on p. 13).

[Ric17a] Chris Richardson. *A Pattern Language for Microservices*. 2017. URL: http://microservices.io/patterns/ (visited on 12/01/2019) (cit. on p. 40).

[Ric17b] Chris Richardson. *Microservices Patterns*. 2017. arXiv: 1-933988-16-9. URL: https://microservices.io/ (cit. on pp. 29, 40, 72, 152).

[Rig17] RightScale. *State of the Cloud Report*. Tech. rep. 2017, p. 38. URL: http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-cloud-survey (cit. on pp. 29, 30).

[Rig19] RightScale. *2019 State of the Cloud Report*. 2019. URL: https://www.flexera.com/2019-cloud-report (visited on 04/01/2020) (cit. on pp. iii, v, 3, 15, 26–28, 33, 44, 217).

[Roc13] James Roche. "Adopting devops practices in quality assurance." In: *Communications of the ACM* 56.11 (2013), pp. 38–43. ISSN: 00010782. DOI: 10.1145/2524713.2524721 (cit. on pp. 19, 25, 31).

[Rol+00] Colette Rolland et al. "Evaluating a pattern approach as an aid for the development of organisational knowledge: An empirical study." In: *International Conference on Advanced Information Systems Engineering*. Springer. 2000, pp. 176–191 (cit. on p. 213).

[Sal00] Nikos A Salingaros. "The structure of pattern languages." In: *ARQ: Architectural Research Quarterly* 4.2 (2000), pp. 149–162 (cit. on p. 213).

[Sas+16] A. Sasabe et al. "Pattern Mining Patterns A Search for the Seeds of Patterns." In: *Pattern Language of Patterns* (2016), pp. 1–16 (cit. on p. 23).

[Sav11] Laura Savu. "Cloud computing: Deployment models, delivery models, risks and research challanges." In: *2011 International Conference on Computer and Management, CAMAN 2011* (2011), pp. 1–4. ISSN: 03605442. DOI: 10.1109/CAMAN.2011.5778816 (cit. on pp. 2, 29, 77).

[Sch+06] Markus Schumacher et al. *Security Patterns: Integrating Security and Systems Engineering*. 2006, p. 600. ISBN: 9780470858844 (cit. on p. 135).

[Sch14] Mathijs Jeroen Scheepers. "Virtualization and Containerization of Application Infrastructure : A Comparison." In: *21st Twente Student Conference on IT* (2014), pp. 1–7 (cit. on p. 78).

[Sei17] Niels Seidel. "Empirical evaluation methods for pattern languages: Sketches, classification, and network analysis." In: *ACM International Conference Proceeding Series*. Vol. Part F1320. 2017, p. 24. ISBN: 9781450348485. DOI: 10.1145/3147704.3147719 (cit. on p. 70).

[Ser15] Amazon Web Services. *Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region*. 2015. URL: https://aws.amazon.com/message/5467D2/ (visited on 12/01/2019) (cit. on p. 32).

[Ser17] Amazon Web Services. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. 2017. URL: https://aws.amazon.com/message/41926/ (visited on 12/01/2019) (cit. on pp. 32, 78).

[Ser19a] Amazon Web Services. *AWS Global Infrastructure*. 2019. URL: https://aws.amazon.com/about-aws/global-infrastructure/ (cit. on p. 17).

[Ser19b] Amazon Web Services. *Regions and Availability Zones*. 2019. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (visited on 12/01/2019) (cit. on p. 31).

[SFC18] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. "Overview of a Pattern Language for Engineering Software for the Cloud." In: *25th Conference on Pattern Languages of Programs*. Portland, Oregon, USA, 2018. ISBN: 9781941652091 (cit. on p. 258).

[SFC20] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Correia. *A Survey on the Adoption of Patterns for Engineering Software for the Cloud - dataset*. Zenodo, 2020. DOI: 10.5281/zenodo.3755046. URL: https://doi.org/10.5281/zenodo.3755046 (cit. on p. 198).

[SFJ96] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. "Software Patterns." In: *Communications of the ACM* 39.10 (1996), pp. 37–39. ISSN: 00010782. DOI: 10.1145/236156.236164 (cit. on p. 6).

[Sha15] Mojtaba Shahin. "Architecting for devops and continuous deployment." In: *ACM International Conference Proceeding Series* 28-Septemb (2015), pp. 147–148. DOI: 10.1145/2811681.2824996 (cit. on p. 31).

[Sim16] Jack Simpson. *What are first, second and third-party data?* 2016. URL: https://econsultancy.com/blog/67674-what-are-first-second-and-third-party-data (cit. on p. 177).

[SM11] Americo Sampaio and Nabor Mendonça. "Uni4Cloud: An approach based on open standards for deployment and management of multi-cloud applications." In: *Proceedings - International Conference on Software Engineering* (2011), pp. 15–21. ISSN: 02705257. DOI: 10.1145/1985500.1985504 (cit. on p. 17).

[SM13] Tiago Boldt Sousa and Angelo Martins. *Monitor, control and process-an adaptive platform for ubiquitous computing*. Vol. 8091 LNCS. September. Alcudia, Mallorca, Spain, 2013, pp. 47–50. ISBN: 9783642408397. DOI: 10.1007/978-3-642-40840-3-7 (cit. on pp. 68, 262).

[SNP15] Jens Smeds, Kristian Nybom, and Ivan Porres. "DevOps: A definition and Perceived Adoption Impediments." In: *Lecture Notes in Business Information Processing* 212 (2015), pp. 166–177. ISSN: 18651348. DOI: 10.1007/978-3-319-18612-2_14 (cit. on p. 19).

[Sol+07] Stephen Soltesz et al. "Container-based operating system virtualization." In: *ACM SIGOPS Operating Systems Review* 41.3 (2007), p. 275. ISSN: 01635980. DOI: 10.1145/1272998.1273025 (cit. on pp. 82, 87).

[Sou12] Tiago Boldt Sousa. "Dataflow Programming: Concept, Languages and Applications." In: *7th Doctoral Symposium on Informatics Engineering*. Vol. 7. Lisbon, Portugal, 2012, p. 13. ISBN: 9789727521418. URL: http://paginas.fe.up.pt/%7B~%7Dprodei/dsie12/papers/paper%7B%5C_%7D17.pdf (cit. on p. 259).

[Sou13] Tiago Boldt Sousa. "Sensors, actuators and services - A distributed approach." In: *SPLASH 2013 - Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity* (2013), pp. 161–165. DOI: 10.1145/2508075.2508464 (cit. on pp. 68, 263).

[Sow87] Henry A. Sowizral. "Design methodology for object-oriented programming OOPSLA'87 panel session." In: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum), OOPSLA 1987* 1987-Janua.October (1987), pp. 91–95 (cit. on p. 21).

[SS17] Ken Schwaber and Jeff Sutherland. "The Scrum Guide: The Definitive The Rules of the Game." In: *Scrum.Org and ScrumInc* November (2017), p. 19. ISSN: 00195847. DOI: 10.1053/j.jrn.2009.08.012. arXiv: arXiv:1011.1669v3 (cit. on p. 18).

[Sta17] Statuscake. *StatusCake*. 2017. URL: https://www.statuscake.com/ (visited on 12/01/2019) (cit. on p. 133).

[Sti15] Matt Stine. *Migrating to Cloud-Native Application Architectures*. O'Reilly Media, Inc., 2015. ISBN: 9780470873939 (cit. on pp. 29, 152).

[Sve17] Yevgeniy Sverdlik. *AWS Outage that Broke the Internet Caused by Mistyped Command*. 2017. URL: https://www.datacenterknowledge.com/archives/2017/03/02/aws-outage-that-broke-the-internet-caused-by-mistyped-command (cit. on p. 31).

[Taf15] Darryl Taft. *How the Skills Gap Is Threatening the Growth of App Economy*. 2015. URL: http://www.eweek.com/developer/slideshows/how-the-skills-gap-is-threatening-the-growth-of-app-economy.html (cit. on p. 3).

[TCB14] Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. "Interconnected Cloud Computing Environments." In: *ACM Computing Surveys* 47.1 (2014), pp. 1–47. ISSN: 03600300. DOI: 10.1145/2593512 (cit. on pp. 2, 14, 29).

[Tec14] Saugatuck Technology. *Why DevOps Matters : Practical Insights on Managing Complex & Continuous Change*. Tech. rep. 2014 (cit. on pp. 19, 31).

[Tei16] Carlos Teixeira. "Towards DevOps: Practices and Patterns from the Portuguese Startup Scene." Master's Thesis. Faculty of Engineering, University of Porto, 2016 (cit. on pp. 62, 64, 66).

[TKP04] E Todd, E Kemp, and Chris Phillips. "What makes a good User Interface pattern language?" In: *Proceedings of the fifth conference on Australasian user interface-Volume 28*. Australian Computer Society, Inc. 2004, pp. 91–100 (cit. on p. 213).

[TSB10] Wei Tek Tsai, Xin Sun, and Janaka Balasooriya. "Service-oriented cloud computing architecture." In: *ITNG2010 - 7th International Conference on Information Technology: New Generations* (2010), pp. 684–689. DOI: 10.1109/ITNG.2010.214 (cit. on pp. 14, 42).

[Uni03] International Telecommunication Union. *Building the Information Society: a global challenge in the new Millennium*. Tech. rep. 2003. URL: https://www.itu.int/net/wsis/docs/geneva/official/dop.html (cit. on p. 1).

[Uni16] The European Parliament Union. *EU Data Protection Rules*. 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1528874672298%7B%5C&%7Duri=CELEX%7B%5C%%7D3A32016R0679 (visited on 12/01/2019) (cit. on pp. 26, 32).

[UX13] Sultan Ullah and Zheng Xuefeng. "Cloud Computing Research Challenges." In: (2013), pp. 1397–1401. arXiv: 1304.3203. URL: http://arxiv.org/abs/1304.3203 (cit. on pp. 28–30).

[VW12] Sitalakshmi Venkatraman and Bimlesh Wadhwa. "Cloud Computing A Research Roadmap in Coalescence with Software Engineering." In: *Software Engineering: An International Journal (SEIJ). Vol. 2. No. 2. September 2012.* 2.2 (2012) (cit. on p. 29).

[WF12] Tim Wellhausen and Andreas Fiesser. "How to write a pattern? A rough guide for first-time pattern authors." In: *ACM International Conference Proceeding Series* (2012), pp. 1–9. DOI: 10.1145/2396716.2396721 (cit. on p. 69).

[Wil12] Bill Wilder. *Cloud Architecture Patterns*. 2012, p. 182. ISBN: 978-1-4493-1977-9. DOI: 10.1007/978-3-642-20917-8 (cit. on p. 72).

[Wil15] Jason Wilder. *Automated Nginx Reverse Proxy for Docker*. 2015. URL: http://jasonwilder.com/blog/2014/03/25/automated-nginx-reverse-proxy-for-docker/ (visited on 12/01/2019) (cit. on p. 147).

[Win99] E. O. Winstedt. "A Bodleian MS. of Juvenal." In: *The Classical Review* 13.4 (1899), pp. 201–205. ISSN: 14643561. DOI: 10.1017/S0009840X00078409 (cit. on p. 133).

[Wol06] Geoffrey H Wold. "Disaster Recovery Planning Process." In: *Disaster Recovery Journal* 5.1 (2006), pp. 1–8 (cit. on p. 33).

[Wug15] Patrick Wuggazer. "Evaluation of an Architecture for a Scaling and Self-Healing Virtualization System." PhD thesis. University of Magdeburg, 2015, pp. 7–35 (cit. on p. 148).

[Xav+13] Miguel G. Xavier et al. "Performance evaluation of container-based virtualization for high performance computing environments." In: *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013* LXC (2013), pp. 233–240. DOI: 10.1109/PDP.2013.41. arXiv: 1709.10140 (cit. on pp. 86, 87).

[Xeb19a] XebiaLabs. *Periodic Table of DevOps (v3)*. 2019. URL: https://xebialabs.com/periodic-table-of-devops-tools/ (visited on 12/01/2019) (cit. on p. 19).

[Xeb19b] XebiaLabs. *The Ultimate DevOps Tool Chest*. 2019. URL: https://xebialabs.com/the-ultimate-devops-tool-chest/ (visited on 12/01/2019) (cit. on pp. 19, 33).

[YH02] Graham Yarbrough and Sandy Hook. *Message Queue Server System*. 2002. URL: https://www.google.com/patents/US20020004835 (cit. on p. 140).

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. "Cloud computing: State-of-the-art and research challenges." In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18. ISSN: 18674828. DOI: 10.1007/s13174-010-0007-6 (cit. on pp. 2, 14, 17).

[Zim+] Olaf Zimmermann et al. *Microservice API Patterns*. URL: https://microservice-api-patterns.org/ (visited on 01/10/2020) (cit. on p. 42).