Overview of a Pattern Language for Engineering Software for the Cloud

Tiago Boldt Sousa, tbs@fe.up.pt, Department of Informatics Engineering, Faculty of Engineering, University of Porto
Hugo Sereno Ferreira, hugosf@fe.up.pt, INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto
Filipe Figueiredo Correia, filipe.correia@fe.up.pt, Department of Informatics Engineering, Faculty of Engineering, University of Porto

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.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**] Patterns

General Terms: Design

Additional Key Words and Phrases: Cloud Computing, Design Patterns, Software Engineering

ACM Reference Format:

Sousa, T.B., Ferreira, H.S., Correia, F.F.. 2018. Overview of a Pattern Language for Engineering Software for the Cloud. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 9 pages.

1. INTRODUCTION

Today, over 49% of the world population has access to at least one Internet enabled device [Internetlivestats.com 2016]. Exploiting this scale with cloud applications made the Internet an appealing channel for running businesses. Such resulted in an explosion in the adoption of public cloud services, with it's market surpassing US$204 billions in 2016 [Woods and Meulen 2016].

Motivated by such widespread of the Internet and the explosive growth of software businesses built on top of it, software engineering became one of the fastest expanding branches of engineering. In fact, the demand for new engineers grows at an higher rate than the pace at which they are graduating [Taft 2015].

Using the cloud as a foundation for application development introduces new challenges and is essential for any Internet business. Still, there is a clear lack of research supporting developers design decisions while for cloud software, namely, identifying what forces drive successful cloud software, and the guidelines to optimize them in order to craft better cloud software.

With this research, the author proposes a pattern language for building cloud software, helping development teams make informed architectural decisions that will improve their software operations. The pattern language

can be used by less experienced teams to bootstrap their design decisions or by experts to validate their own existing architectures. Finally, it provides a framework for reasoning, given that the concepts introduced by the pattern language can be used to argue about cloud architectures using the ontology it introduces.

2. A PATTERN LANGUAGE FOR ENGINEERING SOFTWARE FOR THE CLOUD

This research contributes to the field of software engineering with a pattern language that helps in the design process of cloud software architectures. This section introduces the research questions of this work, presents the pattern language structure and briefly describing each pattern and, finally, demonstrates how it can be applied using an example scenario. The patterns that compose the pattern language are described in **??**.

2.1 Mining the Patterns

While capturing this pattern language, some considerations where taken in order to ensure the individual quality of each pattern. Inspired by the evaluation framework proposed by Seidel [Seidel 2017], the following attributes were considered:

Completness: Is the pattern description complete? [Alexander 1979] A complete pattern provides a level of detail that enables the reader to identify with the problem and reproduce the solution.

Briefness: Does the pattern contain more information than what's 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 accurate examples.

2.2 How To Read These Patterns

The patterns described in this work have all been documented using the pattern structure described below, strongly inspired by classical pattern templates [Meszaros 1998; Wellhausen and Fiesser 2011]. Each pattern is 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 be able to understand what is the driver 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 makes it a complex problem to solve.

Problem: Formalizes the problem, detailing on 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 the implementation of this pattern.

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 intricacies of the pattern.

This structure is a superset of *A Pattern Language for Pattern Writing* [Meszaros 1998], which suggested the common structure of context, problem, forces and solution and it is so structures in order to facilitate the reader's navigations through the pattern, easily identifying or skipping the information he is looking for or is not interested in, respectively.

2.3 Pattern Language Overview

This pattern language has been organized into four pattern categories. This subsection describes each of those categories as well as it briefly describes each pattern's problem and solution.

2.3.1 *Automated Infrastructure Management.* The patterns referenced in this section are not introduced by this pattern language, but a reference to them is essential, as they provide the grounds to much of what this pattern language builds upon. Automated Infrastructure Management if often associated with DevOps and has been extensively researched and has such will remain outside the scope of this research [Cycligent ; Loukides 2012; Reed 2014; XebiaLabs ; Technology 2014; Erich et al. 2014].

Infrastructure as Code is one of the most relevant contributions from the DevOps mindset to software engineering for the cloud. This pattern handles the management of infrastructure using software [Smeds et al. 2015; Hüttermann 2012; Riley 2015; Amazon 2018]. For that, the creation and changes to the infrastructure are defined during the development process, following the development practices in place by the team. This will usually ensure the quality of the software created to manage the infrastructure through practices such has peer review. Furthermore, Infrastructure as Code, following the DevOps mindset, deprecates the need for knowledge operation workers that would be dedicated to setup and handle infrastructure. The team owns that responsibility, managing it as part of the development cycle. This automation will further increase confidence in changes, as it eliminates the possibility of human introduced error.

Another known pattern required by this pattern language, this time available from cloud providers, is Automated Scalability. This pattern enables a cloud provider to monitor a set of instances for their resource allocation, scaling them appropriately to balance performance and costs [Zhang et al. 2010; Cycligent ; Fehling et al. 2014]. As an example of how this pattern works, an infrastructure with ten instances that is experiencing increased CPU usage can be automatically scaled, one machine at the time, until the average CPU load in the infrastructure is below a configured maximum threshold. On the other hand, if the average CPU usage is below a minimum threshold, the infrastructure size can be reduced, one machine at the time, until the average CPU load it above a desired average threshold. This will ensure that the infrastructure is maintains an overall CPU usage within the desired threshold interval.

2.3.2 *Orchestration and Supervision.* Infrastructure empowering software in the cloud is typically volatile and dynamically allocated. As such, orchestration plays a key role at dynamically identifying the execution setup and adapt the software to cope with it. The patterns in this section describe how to setup the necessary hardware and software to orchestrate services in the cloud and insure they run flawlessly.

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 get scattered in the host making managing the machine and hosting multiple applications troublesome. While virtualization can be used to 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 avoid the need to install dependencies and have configurations

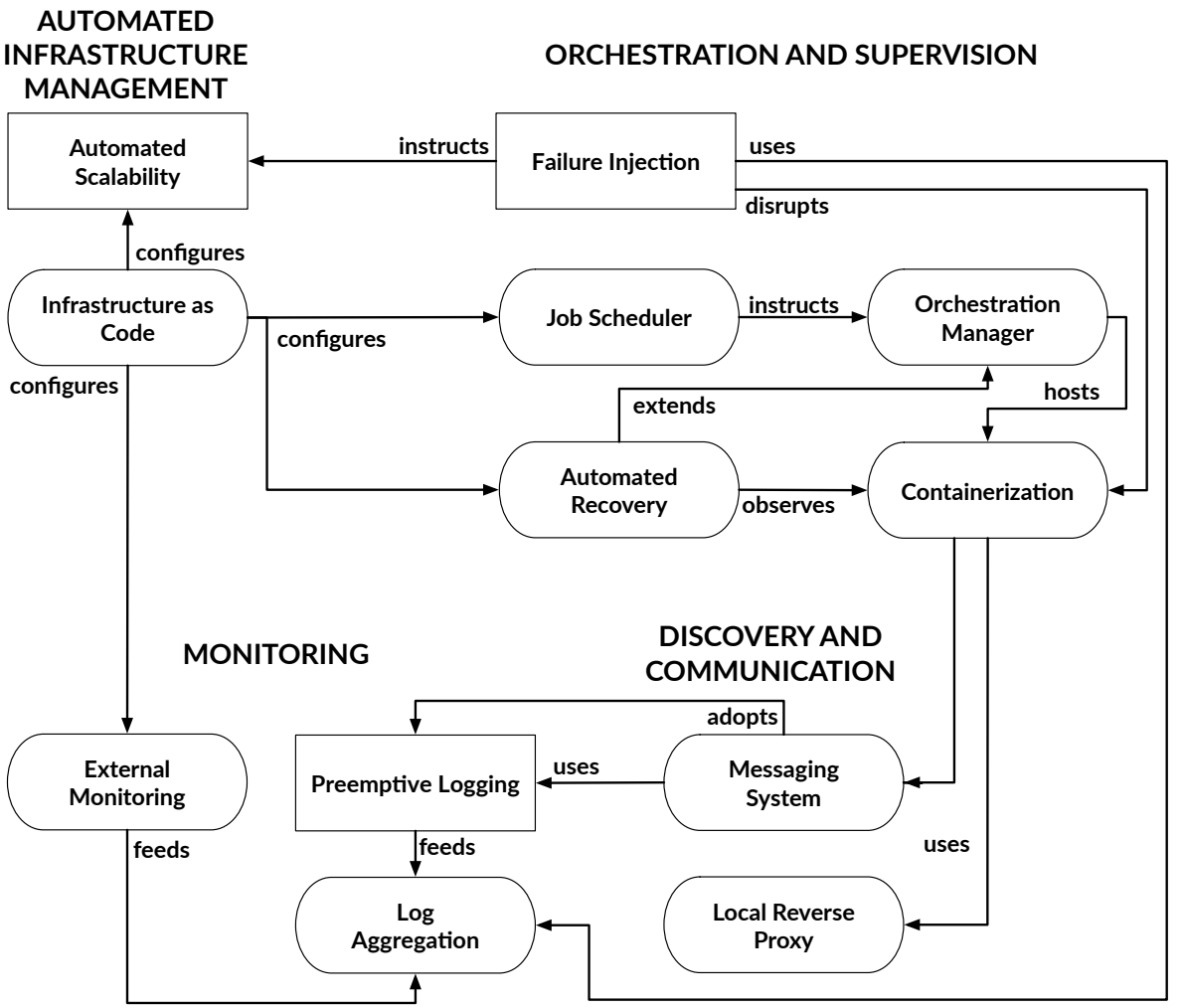

Fig. 1: Pattern map for engineering software for the cloud. Patterns are identified as architecture or process patterns and aggregated into one of the four categories. Arrows identify the relationships between the patterns. Rounded rectangles correspond to architectural patterns, while plain rectangles are process patterns.

scattered within the host, making them easier to manage and deploy at scale [Boldt Sousa et al. 2015; Scheepers 2014]. Having at most one service per container ensures that all services are isolated from each-other, preventing secondary effects in their behaviors. This approach is also essential for individually scaling each microservice. Adopting containers facilitates the service's portability. Configurations should not exist inside the container, but instead services should acquire their configuration from environment variables made available to the container in each specific execution environments. Environment-based configuration enables not only the service's configuration to be injected into the environment, but the host resources' details to be made available in the same way.

Servers in an infrastructure might differ in hardware. While some might provide more CPU, others might have higher amounts of RAM available. Likewise, services requirements also are unique. While some might require a specific amount of memory to be available, other might have the need to be co-located in the same host for latency purposes. As such, services need to be co-located within hardware fulfills their resource and logical

requirements. An Orchestration Manager should be responsible for allocating services to the proper hosts, considering their overall and available resources, as well as any other allocation restrictions that the service might have [Boldt Sousa et al. 2015; Odewahn 2014; Hausenblas]. This pattern works best when services are distributed using the already described Containerization pattern.

Asynchronous tasks, such as database maintenance, sending emails or performing backups, are common in cloud software. 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 tasks in an infrastructure and evaluate their result, generating error reports when need [Boldt Sousa et al. 2018b].

Software fails. 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 [Boldt Sousa et al. 2018b].

Cloud Software has typically a desired uptime of 100%. Redundancy is often a first step to guarantee availability. Still, it is a fact that software fails [Charette 2005]. As such, developers need to ensure that their application is resilient to failures and able to recover to a working status without human intervention, with mechanisms for improving resilience being built into the cloud application. To increase confidence in these mechanisms they need to be themselves tested. And since the environment might influence how the system behaves, the production system's resilience itself must be tested. To do so a Failure Injection mechanism can periodically inject unexpected events in the system, evaluating if it continues to behave appropriately [Boldt Sousa et al. 2018a]. Such events can range from sending invalid inputs to the application to shutting down one of the servers hosting the application. In both scenarios, if there is any unexpected impact into the application, the resilience mechanisms should be triggered and the application have little to no impact in its availability. Whenever that doesn't happen, the developers can work on improving the resilience behavior. Failure Injection is commonly executed under close supervision and in redundant systems to prevent actual damage to the application.

In summary, the following patterns are introduced in this section:

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.

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.

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 monitor and recover failing containers.

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.

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.

2.3.3 *Monitoring.* Monitoring continuously evaluates the status of the services composing a cloud application, reporting back to the team when an issue is identified. It works reactively by detecting issues on data produced by the services, using log entries or alarms as inputs, or proactively by interacting with the services to verify their status.

While designing the application, all public endpoints and their expected behavior should be identified, so that an EXTERNAL MONITORING tool can be configured to monitor them. This enables the unbiased observation of the application's state from outside its infrastructure, ensuring that the monitoring tool observes the same as an user would [Boldt Sousa et al. 2018a].

Debugging an application in production requires as much information as possible in order to trace the actions that lead to an issue. Services should LOG AGGREGATION, producing verbose execution logs that should be kept for the longest period of time possible [Boldt Sousa et al. 2017].

Having distributed services producing logs will required developers to leverage multiple log files to trace an issue. To prevent this, the team should adopt LOG AGGREGATION, by having a centralized view of all the logs generated by all services in a queryable format [Boldt Sousa et al. 2017].

The following patterns are introduced in this section:

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.

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.

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.

2.3.4 *Discovery and Communication.* Most cloud applications are composed by multiple services cooperating towards providing the whole application in what it is typically called a microservice architecture [Lewis and Fowler 2014]. Being deployed in containers hosted in dynamically provisioned hardware, the services must first discover and create a communication channel before they can start to cooperate.

While using an ORCHESTRATION MANAGER that dynamically allocates containers, the exact network location at where a service will be running is unknown. Using a LOCAL REVERSE PROXY, a service 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 [Schumacher et al. 2006; Boldt Sousa et al. 2015]. This is easily achieve by preemptively creating a table that maps local ports to services. Whenever the port is mapped, the service is up and the communication can be established.

Some use cases require services to communicate amongst themselves synchronously for RPC and asynchronously for delegating information to collaborating services. A MESSAGING SYSTEM can be used to send both types of messages between micro-services, eliminating the complexity associated with service discovery [Gawlick 2002; Boldt Sousa et al. 2017].

This section introduced the following sections:

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.

Local Reverse Proxy: Services might lack the network information required to communicate with other dynamically-allocated services. Communication can be achieved by abstract service network details by defining a service port for each service. Use a reverse proxy to expose that port in all servers, forwarding traffic to where the services are deployed.

3. ADOPTING THE PATTERN LANGUAGE

Resistance to change is by itself a pattern [Dent and Goldberg 1999]. Adopting a pattern language for developing software for the Cloud requires teams to adapt their mindset regarding their organization, processes and software architectures. While it is imperative that the team is motivated to change, this pattern language eases its adoption as it can be partially implemented. The team can identify its most critical problem and implement the pattern or set of patterns that solve it without addressing the whole pattern language. The next section demonstrates how a development team could gradually adopt the described patterns in a sequence of iterations.

Amongst the more experienced users, the pattern language can be used as a ontology for reasoning about cloud architectures, as well as a validation tool for their already existing architectures. Using it as reference will also facilitate how experts share their ideas with less experienced engineers.

This section was inspired by *The Unfolding of a Japanese Tea Garden* by Christopher Alexander [Alexander 2006]. It uses a sequence as a way to help the reader understand how the patterns relate and complement each other. Sequences describe a set of actions that should follow each other in order to achieve a specific goal.

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.

Using INFRASTRUCTURE AS CODE the *practitioner* would describe the initial infrastructure required to setup the system. By executing this programmatic description, the required infrastructure would become available. AUTOMATED SCALABILITY could be setup to ensure that the hardware where the web server executes would scale horizontally if needed according to the provided scalability rules.

To deploy his services in an isolated and scalable way, the infrastructure would be abstracted thought an ORCHESTRATION MANAGER, which 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.

To facilitate discovery in the dynamically allocated hardware, the web server would use the local network port 12345 to connect to the database. Such would be possible given a LOCAL REVERSE PROXY configured in all machines that would expose a static service port for each service instantiated. This scenario would not require the MESSAGING SYSTEM.

To ensure the service is working properly, the *practitioner* would implement the following monitoring techniques.

An EXTERNAL MONITORING service can monitor all public application endpoints, ensuring that they are both online and responding appropriately.

To further increase awareness over the system's state, PREEMPTIVE LOGGING could be adopted to configure the developed and adopted services to use an appropriate level of logging to make the required historical information available to the team to develop issues after they have happened. LOG AGGREGATION can bring all this information to a centralized, indexed and queryable location for easier use.

Finally, and in order to validate the resilience in the system, FAILURE INJECTION can exercise the resilience mechanisms by randomly introducing errors in the infrastructure, such as randomly shutting down machines, and verifying that the system recovers automatically.

4. SUMMARY AND FUTURE WORK

This paper describes a pattern language for engineering software for the cloud. The language is divided into four categories: Automated Infrastructure Management, which references known DevOps patterns upon which this work build; Orchestration and Supervision, proving strategies to abstract an infrastructure's resources and automate service placement within it; Monitoring, which helps the team be aware of the system's status and Discovery and Communication which facilitates inter-service communication within the infrastructure.

This language aims practitioners to make informed decisions while designing their cloud environments, facilitating the creation of resilient and reliable cloud applications.

Plans for future work are twofold. First, continue increasing the completeness of this pattern language by increasing the number of patterns the compose it. Second, increase the patterns quality by surveying experts to gather their input on each individual pattern's attributes: completeness, briefness and validity.

5. ACKNOWLEDGEMENTS

The authors would like to acknowledge Robert Hanmer for his feedback while shepherding this paper.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013».

REFERENCES

Christopher Alexander. 1979. The Timeless Way of Building. *New York Oxford University Press* (1979).

Christopher Alexander. 2006. *The Nature of Order: The Process of Creating Life*. Center for Environmental Structure. 636 pages.

Amazon. 2018. AWS Auto Scaling. (2018). `https://aws.amazon.com/autoscaling/`

Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. 2015. Patterns for Software Orchestration on the Cloud. In *Proceedings of the 2015 Conference on Pattern Languages of Programs*.

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2017. Engineering Software for the Cloud: Messaging Systems and Logging. *Proceedings of the 22Nd European Conference on Pattern Languages of Programs* (2017). `DOI:http://dx.doi.org/10.1145/3147704.3147720`

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018a. Engineering Software for the Cloud : External Monitoring and Fault Injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–13.

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018b. Engineering Software for the Cloud: Automated Recovery and Scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–13.

Robert N. Charette. 2005. Why Software Fails. (2005). `http://spectrum.ieee.org/computing/software/why-software-fails`

Cycligent. *Continuous Delivery Patterns for Design & Deployment*. Technical Report.

Eric B Dent and Susan Galloway Goldberg. 1999. Challenging " Resistance to Change ". 35, 1 (1999), 25–41. `DOI:http://dx.doi.org/10.1177/0021886399351003`

Floris Erich, Chintan Amrit, and Maya Daneva. 2014. Report : DevOps Literature Review. (2014).

Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns*. 239–286 pages. `DOI:http://dx.doi.org/10.1007/978-3-7091-1568-8`

Dieter Gawlick. 2002. Message Queuing for Business Integration. *{eAI} Journal* October 2002 (2002), 30–33.

Michael Hausenblas. *Docker-Networking-and-Service-Delivery*.

Michael Hüttermann. 2012. Infrastructure as Code. In *DevOps for Developers*. Apress, Berkeley, CA, 135–156. DOI:http://dx.doi.org/10.1007/978-1-4302-4570-4{_}9

Internetlivestats.com. 2016. Number of Internet users in the world. (2016). http://www.internetlivestats.com/internet-users/

James Lewis and Martin Fowler. 2014. Microservices. (2014). http://martinfowler.com/articles/microservices.html

Mike Loukides. 2012. *What Is DevOps?* O'Reilly Media. 15 pages. http://shop.oreilly.com/product/0636920026822.do

G Meszaros. 1998. A pattern language for pattern writing. *Pattern languages of program design* (1998). http://xunitpatterns.com/~gerard/plopd3-pattern-writing-patterns-paper.pdfpapers2://publication/uuid/420266E9-5BD8-41A3-AA9B-F03763E9E78E

Andrew Odewahn. 2014. *Field Guide To The Distributed Development Stack*. Vol. 53. 160 pages. DOI:http://dx.doi.org/10.1017/CBO9781107415324.004

J Paul Reed. 2014. *DevOps in Practice*. 34 pages. DOI:http://dx.doi.org/10.1017/CBO9781107415324.004

Chris Riley. 2015. Version Your Infrastructure. (2015). https://devops.com/version-your-infrastructure/http://devops.com/2015/11/12/version-your-infrastructure/

MJ Scheepers. 2014. Virtualization and Containerization of Application Infrastructure: A Comparison. (2014). http://referaat.cs.utwente.nl/conference/21/paper/7449/virtualization-and-containerization-of-application-infrastructure-a-comparison.pdf

Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns: Integrating Security and Systems Engineering*. 600 pages.

Niels Seidel. 2017. Empirical Evaluation Methods for Pattern Languages: Sketches, Classification, and Network Analysis. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. 24.

Jens Smeds, Kristian Nybom, and Ivan Porres. 2015. DevOps: A definition and Perceived Adoption Impediments. *Lecture Notes in Business Information Processing* 212 (2015), 166–177. DOI:http://dx.doi.org/10.1007/978-3-319-18612-2{_}14

Darryl Taft. 2015. How the Skills Gap Is Threatening the Growth of App Economy. (2015). http://www.eweek.com/developer/slideshows/how-the-skills-gap-is-threatening-the-growth-of-app-economy.html

Saugatuck Technology. 2014. *Why DevOps Matters : Practical Insights on Managing Complex & Continuous Change*. Technical Report.

Tim Wellhausen and Andreas Fiesser. 2011. How to write a pattern? *Proceedings of the 16th European Conference on Pattern Languages of Programs - EuroPLoP '11* (2011), 1–9. DOI:http://dx.doi.org/10.1145/2396716.2396721

Viveca Woods and Rob Meulen. 2016. Gartner Says Worldwide Public Cloud Services Market Is Forecast to Reach $204 Billion in 2016. (2016). http://www.gartner.com/newsroom/id/3188817

XebiaLabs. Periodic Table of DevOps. (????). https://xebialabs.com/periodic-table-of-devops-tools/

Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1 (4 2010), 7–18. DOI:http://dx.doi.org/10.1007/s13174-010-0007-6