TEKNISKA HÖGSKOLAN

HÖGSKOLAN I JÖNKÖPING

Towards cloud application architectural patterns: transfer, evolution, innovation and oblivion

Dimitar Dimitrov

EXAM WORK 2014

Information Engineering and Management

Postadress:
Box 1026
551 11 Jönköping

Besöksadress:
Gjuterigatan 5

Telefon:
036-10 10 00 (vx)

This exam work has been carried out at the School of Engineering in Jönköping in the subject area Information Engineering and Management. The work is a part of the Master of Science programme.
The authors take full responsibility for opinions, conclusions and findings presented.

Examiner: He Tan

Supervisor: Rob Day

Scope: 30 credits (second cycle)

Date:

Abstract

Recently, cloud computing has been gaining more and more popularity. Misunderstanding, misusing and underutilizing the cloud offerings, though, both from business and technical perspective still poses a threat to the success of cloud projects. On the technical side, one of the main reasons for success or failure is often the architectural design of the system – if a system is not architected the "cloud way", using cloud's special characteristics, the business benefits of such a system are often questionable at best. Software architecture through architectural patterns – reusable generic solutions to classes of problems – has for long been a good way to overcome the challenges of software architecture.

This paper focuses on establishing the grounds and highlighting the differences of the knowledge transfer regarding architectural patterns from building pre-cloud ("traditional") software systems to building cloud-native systems.

The following 3 research questions drive this research:

RQ1. How does the existing knowledge on architectural patterns relate to the cloud computing environment?

RQ2. Which characteristics of architectural patterns make them suitable for the cloud environment?

RQ3. How can architectural pattern evolution be documented effectively for usage in the practice?

In order to answer these 3 research questions and considering their focus is on utility i.e. creating a model which can be directly used in practice, the research uses design science research methodology (Peffers, et al., 2007-8). The emphasis in this methodology is iteratively building artefact(s) which can be improved and proven through practice that they actually help solving the problem at hand.

This research contributes with building 4 inter-connected artefacts:

- a cloud applicability taxonomy of architectural patterns (CATAP) showing how applicable to a cloud environment an architectural pattern is and why;
- a pattern-to-characteristics mapping showing how using an architectural pattern affects the resulting system in traditional and cloud environments;
- a pattern form documenting the architectural patterns and the findings about them in the previous two artefacts;
- a wiki site, APE Wiki, which makes the results available to the public for reference and discussion and improvement.

This research has a few interesting findings. First of all, the current architectural pattern knowledge seems to be very mature as no pattern has been found to have significantly evolved because of cloud – the architectural patterns are really generic and very flexible and only their effect on system characteristics has changed with the environment switch. On the other hand, a few new patterns were discovered

and documented, which confirms the need for special attention to the new environment. Apart from that, the pattern-to-characteristics mapping provides interesting insights into which characteristics are most important for cloud and where there is a gap which may need to be filled.

This paper presents both the process and the results of the research as equally important as replicating and extending this research could help in maturing the results and the knowledge about architecting systems for cloud thus increasing the chances of success of cloud projects.

Keywords

Architectural Patterns – Cloud Computing – Knowledge Transfer – Enterprise Software Architectures – Software Engineering

Acknowledgements

I would like to express my gratitude to my supervisor, Mr Rob Day, who had the patience and willingness to help me throughout the whole research process. His valuable feedback helped not only the research, but me as a person and professional.

I would also like to thank Mr. Eoin Woods who readily joined the initial effort of defining the scope of the thesis and provided comments which later helped ensure better validity of the results.

Although not directly related, I am very thankful to all students and teachers at Jönköping University who I had the chance to interact with.

Last but not least, I would like to thank my family and friends for the support. I dedicate this work to my mother who had supported me during all my education.

Contents

List of abbreviations i

List of tables ii

List of figures iii

1 Introduction 1

1.1 Background 2
1.2 Purpose and research questions 4
1.3 Delimitations 5
1.4 Conventions 6
1.5 Outline 6

2 Theoretical Background 7

2.1 Important definitions 7
2.1.1 Cloud computing 7
2.1.2 Software architecture 10
2.1.3 Patterns 11
2.1.4 Pattern evolution 12
2.1.5 Frameworks, architectural styles, and architectural tactics 13
2.2 Related work 14
2.2.1 Patterns and pattern languages 14
2.2.2 Cloud application architectural patterns 16
2.2.3 Architectural patterns and architectural decisions 16
2.2.4 Documenting patterns 17
2.2.5 Pattern-oriented system design and evolution through pattern evolution 18

3 Research Method 21

3.1 Choice of method 21
3.2 Design science research 21
3.3 Implementation 24
3.3.1 Problem identification and motivation 24
3.3.2 Definition of solution objectives 25
3.3.3 Design and development 26
3.3.4 Demonstration 29
3.3.5 Evaluation 29
3.3.6 Communication 29

4 Design and Development 30

4.1 Cloud applicability taxonomy of architectural patterns (CATAP) 30
4.1.1 CATAP construction process 30
4.1.2 Selection of architectural patterns for the CATAP 31
4.1.3 The CATAP 34
4.1.4 Discussion of patterns with respect to the CATAP 35
4.1.5 Analysis of the CATAP 39
4.2 Patterns-to-characteristics mapping 41
4.2.1 Selection of characteristics 41
4.2.2 The patterns-to-characteristics mapping 43
4.2.3 Implications for a sample of architectural patterns 44
4.2.4 Analysis 46
4.3 Pattern form 48
4.3.1 Requirements 48
4.3.2 Discussion of options 49

4.3.3 A full pattern form (FPF) *50*
4.3.4 A cloud abbreviated pattern form (CAPF) *52*
4.5 APE WIKI 53
4.5.1 Requirements *53*
4.5.2 Discussion of options *54*
4.5.3 Implementation *54*

5 Demonstration 56

5.1 DEMONSTRATION IN THIS RESEARCH 56
5.2 OPTIONS FOR EXTENSION 56

6 Evaluation 58

6.1 EVALUATION IN THIS RESEARCH 58
6.1.1 Evaluation based on the research objectives *58*
6.1.2 Evaluation of reliability of the results *59*
6.1.3 Extensibility and replication of the research process *60*
6.2 OPTIONS FOR EXTENSION 60

7 Discussion and Conclusions 61

7.1 DISCUSSION OF RESEARCH METHOD 61
7.2 DISCUSSION OF FINDINGS 62
7.3 OPTIONS FOR FUTURE RESEARCH 62

References 63

Terms and Definitions 70

Search Terms 76

Appendices 78

APPENDIX 1: CLOUD APPLICABILITY TAXONOMY OF ARCHITECTURAL PATTERNS (CATAP) 79
APPENDIX 2: RELEVANCE MATRIX FOR THE CHARACTERISTICS DEFINED IN (ISO/IEC 25010:2011, 2011B) 83
APPENDIX 3: PATTERNS-TO-CHARACTERISTICS MAPPING 88
APPENDIX 4: LAYERS PATTERN DOCUMENTED USING ARTEFACT 3, A PATTERN FORM 89

List of abbreviations

APE Wiki	Architectural Patterns' Evolution Wiki
CAAP	Cloud Application Architectural Pattern
CAPF	Cloud Abbreviated Pattern Form
CATAP	Cloud Applicability Taxonomy of Architectural Patterns
CIAP	Cloud Infrastructural Architectural Pattern
CQRS	Command Query Responsibility Segregation
DSRM	Design Science Research Methodology
FPF	Full Pattern Form
IaaS	Infrastructure as a Service
ISO	International Organization for Standardization
IT	Information Technology
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
SaaS	Software as a Service
SEI	Software Engineering Institute
SLARF	Scalability, Location transparency, Availability, Recoverability, Fault-tolerance

List of tables

TABLE 1 DESIGN SCIENCE RESEARCH GUIDELINES (HEVNER, ET AL., 2004) 23
TABLE 2 PATTERNS-TO-CHARACTERISTICS MAPPING (EXCERPT) 43
TABLE 3 EVALUATION MATRIX FOR WIKI ENGINES. 54

List of figures

FIGURE 1 CLOUD SERVICE MODELS (CIO RESEARCH CENTER, 2010). 9
FIGURE 2 DESIGN SCIENCE RESEARCH CYCLES (HEVNER & CHATTERJEE, 2010B). 22
FIGURE 3 DSRM PROCESS MODEL (PEFFERS, ET AL., 2007-8). 22
FIGURE 4 THE THESIS RESEARCH METHODOLOGY BASED ON PEFFERS, ET AL. (2007-8). 24
FIGURE 5 RELATIONS BETWEEN RESEARCH QUESTIONS, ARTEFACTS AND GOAL. 27
FIGURE 6 TRADITIONAL ARCHITECTURAL PATTERNS – A PATTERN LANGUAGE FROM (AVGERIOU & ZDUN, 2005). 32
FIGURE 7 CLOUD APPLICABILITY TAXONOMY OF ARCHITECTURAL PATTERNS (CATAP). 35
FIGURE 8 CQRS. SOURCE: (FOWLER, 2011). 39
FIGURE 9 PATTERN SUPPORT FOR CHARACTERISTICS BY CLOUD APPLICABILITY CATEGORIES. 46
FIGURE 10 A FULL PATTERN FORM (FPF) 51
FIGURE 11 ARTEFACT 3, A CLOUD ABBREVIATED PATTERN FORM (CAPF) 52
FIGURE 12 APE WIKI: LEFT HAND SIDE NAVIGATION 55

1 Introduction

Cloud computing has emerged in the last few years as an important direction in the Information Technology (IT) area. In essence, IT resources, such as servers, storage, networking, and software tools and platforms, are rented out as utility or a service over the Internet (Guha, 2013, p. 9). Consumers of cloud technologies can avoid the complexity of managing the underlying infrastructure and are provided with tools to control and utilize the resources they rent. This cloud computing model and the environment it creates have distinct characteristics compared to what is called traditional model and traditional environment in this paper.

The traditional model encompasses 2 options: (1) to host a system on owned hardware which is usually referred to as on-premises or in-house hosted solution, or (2) to use a hosting provider which provides a predefined amount of hardware and software resources either dedicated, or shared with other consumers. Either way, cloud computing possesses very distinctive characteristics compared to the traditional model – for example, a system can be dynamically scaled based on its usage. These characteristics sometimes cause very significant differences between the cloud and the traditional model; these differences affect the whole IT area including the field of software architectures (Krishna & Jayakrishnan, 2013, p. 83).

Many enterprises have already built systems for the cloud environment, but many more are still wondering on whether and how to take advantage of the various cloud offerings. In order to utilize cloud computing, enterprises have 2 distinct options at hand – to change and re-target an existing system or to have a new system built especially for cloud. Either way, research shows that questionable or even counter-effective results are achieved by simply deploying an existing system to a cloud environment without making changes such that the system makes use of cloud offerings and respects the particularities of the environment (Fehling, et al., 2011).

Architecting through the use of patterns is one of the significant directions in the field of software architectures. Architectural patterns capture practical knowledge and give a general template solution on how to cope with software architectural design problems in a specific context.

Having in mind that cloud computing is a new context that requires new ways of architecting software, changes in the architectural patterns domain are very probable. The research presented here concentrates on these changes investigating the applicability of existing patterns, their evolution to respond to the changed environment and the emergence of completely new patterns. The ultimate goal is to enable practitioners to transfer more effectively their existing architectural knowledge to cloud computing projects.

This paper is written as a Master thesis work. It is a part of the Master of Science program of Information Engineering and Management at the School of Engineering, Jönköping University, Sweden.

1.1 Background

In 1977 Christopher Alexander pioneered the term "pattern" in his book *A pattern language* (Alexander, et al., 1977). The book is in his area of specialty – building architecture. A pattern according to this definition "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, et al., 1977). Quite surprisingly for Alexander himself (Gabriel, 1996, p. v), the concept of patterns has caused a significant impact in a completely different field – software engineering.

The process of pattern adoption in software engineering took off with Ward Cunningham and Kent Beck's *Constructing Abstractions for Object-Oriented Applications* (Cunningham & Beck, 1987). A strong community of researchers and practitioners supporting the idea of software patterns has been formed, pushing the field forward. The book *Design patterns elements of reusable object-oriented software* (Gamma, et al., 1995) further motivated and enabled extended usage in the practice. In this book the patterns "live" at the code-level and are looked at from an object-oriented perspective – they deal with structuring, organizing and developing a class or a hierarchy of classes which present a solution with the properties Alexander (1977) talks about. These patterns are now commonly known in the software engineering world as design patterns.

With IT penetrating more and more aspects of people's lives, software systems are growing in complexity. The need has become evident for a more deliberate process in which design decisions lead to more predictable results. This has led to the emergence of the software architecture discipline – a new discipline focused on software architectures. A concise and practical definition states that "software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled" (Woods, 2014).

Patterns as solutions with well-known characteristics for a specific class of problems in a specific context are a good candidate for helping in achieving the goals of the software architecture discipline – this is the idea behind research and documentation of architectural patterns. Patterns of this type express "a fundamental structural organization schema for software systems" (Buschmann, et al., 1996, p. 12). Architectural patterns often reuse the knowledge built from design patterns, but the system perspective also brings new problems to be solved which gives rise to new approaches and solutions.

As with design patterns, the area of architectural patterns has been actively researched and documented. Classic examples of books on the subject are *Patterns of Enterprise Application Architecture* (Fowler, et al., 2002), *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Hohpe & Woolf, 2003), the *Pattern-Oriented Software Architecture* series (Buschmann, et al., 1996; Schmidt, et al., 2000; Kircher & Jain, 2004; Buschmann, et al., 2007a; Buschmann, et al., 2007b).

Technological and business advances, together with the quest for more effective and efficient work processes, have steadily pushed the field of software engineering. This dynamic causes constant and sometimes substantial changes and the problems in front of the field constantly change as well. As patterns are dependent on the design context they are used in (Buschmann, et al., 1996, p. 8), it is natural for some of them to transfer to the new environment, others to evolve and be adapted for the new conditions, new patterns to be invented and old ones to reach a lower level of relevance. This paper investigates exactly this change in the architectural patterns in the context of moving from "traditional" software engineering to software engineering in the cloud.

In this thesis "traditional", as applied to software engineering or patterns, is used solely to signify the pre-cloud context. In that sense, "traditional" encompasses engineering of software systems for deployment on owned hardware or for deployment to a traditional hosting provider; the former deployment approach is often also called on-premises or in-house. In order to explain the differences between the traditional and the cloud model, the following widely adopted definition by the National Institute of Standards and Technology (NIST) should be discussed:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models (NIST: Mell & Grance, 2011)

In order to draw the boundary between traditional on-premises and cloud environment, it is important to highlight from the definition above the fact that cloud computing is supplied by a cloud provider. This means cloud systems are hosted externally for the consumer – this observation makes the difference between on-premises and cloud hosted systems rather clear as on-premises solutions are hosted on owned IT infrastructure internally at the customer.

The differences between cloud hosted systems and systems hosted by a traditional hosting provider are not that obvious, though. Again, pieces of the definition above help in unveiling the differences: cloud enables "on-demand network access to a shared pool of configurable computing resources […] that can be rapidly provisioned and released with minimal management effort or service provider interaction" (NIST: Mell & Grance, 2011). First of all, in traditional hosting there are usually packages of coarse-grained infrastructure that is provided to a consumer. These packages are usually static, not dependent on concrete usage and require communication with the hosting provider in order to be switched – these characteristics are all in contrast with what is stated and typical for cloud environments. Second, the provider in the cloud definition is called "service provider" – services available and accessible over the Internet (making them "ubiquitous") are a major characteristic of cloud environments, whereas in

traditional hosting IT infrastructural resources i.e. dedicated or shared servers, are the main focus. This second point also hints at a difference in favour of cloud in terms of management effort – provisioning and maintenance of resources should require minimal effort for cloud, whereas in traditional hosting the consumer needs to take care of the resources borrowed which is even comparable to on-premises environments at least on the software maintenance front. As a third point, because of the economy of scale, typically the cost of using cloud resources is lower than the cost of using a traditional hosting provider. And finally, the economy of scale and the better utilization of resources because of the rapid provisioning and releasing of resources based on usage allow for providing increased availability out of the box through ensuring redundancy in cloud environments. The importance of that characteristic is also ensured usually through a Service Level Agreement, which guarantees high availability for cloud hosted systems; although traditional hosting also provides availability promises and guarantees, they are usually not that strong.

After the definition helped draw the boundary between traditional and cloud environments, there is still a part of it that remains undiscussed. The characteristics and models mentioned in the last sentence of the definition are discussed in detail in section 2, Theoretical Background.

1.2 Purpose and research questions

As stated by Marinescu (2013) cloud computing "is expected to shift from building the infrastructure, today's main front of competition among the vendors, to the application domain". This means that the accent will fall once more on the development of systems and the best way of architecting them. So it is of utmost importance to adapt and transfer the existing architectural knowledge in accordance with the cloud characteristics and context.

Both software architecture as a discipline aimed to support system design and architectural patterns as an important concept in the field need reconsideration when applied to cloud computing. It is important to know what the changes are and why they have happened in order to use and transfer effectively the existing architectural knowledge to the new environment. In order to achieve that transfer this paper concentrates on investigating the influences of the cloud context to architectural patterns. Four cloud applicability categories with respect to these changes were introduced above and are highlighted in the title of the paper:

- Transfer – the architectural patterns that move to the cloud without changes;
- Evolution – those patterns which are adapted to support cloud usage;
- Innovation – completely new patterns that have emerged for the cloud specifically;
- Oblivion – the category of architectural patterns that are not that relevant/suitable for cloud usage.

Further, without considering the cloud characteristics and the architectural principles applicable for the cloud environment, one cannot fully realize the potential offered by cloud (Fehling, et al., 2011).

All this motivates the ultimate goal of the thesis - to enable practitioners to transfer more effectively their existing architectural patterns knowledge to cloud computing projects. With respect to it the following questions were formulated:

RQ1. How does the existing knowledge on architectural patterns relate to the cloud computing environment?

RQ2. Which characteristics of architectural patterns make them suitable for the cloud environment?

RQ3. How can architectural pattern evolution be documented effectively for usage in the practice?

These questions are inspired and are related to the four artefacts constructed during the thesis: (1) a cloud applicability taxonomy of architectural patterns (abbreviated to CATAP), (2) a patterns-to-characteristics mapping, (3) a documentation template, and (4) a wiki, code-named APE Wiki (Architectural Patterns' Evolution Wiki). The section Research Method covers these artefacts in detail, together with the relationships between them and how they relate to the research questions.

1.3 Delimitations

This paper is written as a Master thesis work which has to be performed within a one calendar year period. This therefore limits the possible scope of the work. The thesis concentrates on detailing the research process and defining and testing a method for achieving the goal of the research. Therefore, as the focus is not on extensive coverage of the field, a subset of the hundreds known architectural patterns is examined in depth. The observations made over this subset helped refine the process. This report documents the research process so that it can be repeated and possibly further refined.

Architectural design is a complex decision process. The results outlined in this paper are intended to provide support in knowledge transfer, knowledge modelling and evaluation of architectural design alternatives. They should not be used instead of taking conscious design decisions.

The artefacts designed and developed in the scope of the paper (models, prototypes, information systems) are used for illustration purposes, evaluation and conveying the findings. Indeed some of the artefacts may not be complete and require further development – an initial attempt is done within the paper, but refinement in that case is best done by the community of practitioners and this requires time. This is in compliance with the research method described in the section Research Method, which recognizes that a number of iterations are needed for refinement of results.

1.4 Conventions

In this paper several conventions are used. These are:

- All pattern names are written in UPPERCASE in order to make them stand out.
- All book names are in *italic*.
- Where an important term is introduced for the first time it is in ***bold and italic***

Also the term ***cloud***, where used alone, is used as an all-encompassing term referring to the cloud environment considering all its aspects and the related cloud characteristics.

As far as source referencing is concerned, some books referenced were retrieved in digital format and the exact printed pages which the references were extracted from could not be retrieved. In all other cases, the information about the pages of origin is included in the reference.

Throughout this paper British English spelling is preferred. The exception to this preference are verbatim cites – the original, most often American English, spelling is preserved. An example is the spelling of the word "artefact" (British English) – in American English it is spelled "artifact". The first appearance of such a word is denoted with a footnote and the alternative spellings are provided.

1.5 Outline

The rest of the report is organized as follows. In section 2, Theoretical Background, the base of the research is established in terms of important definitions, review of the related literature and the trends in the research area. The section Research Method describes the chosen research method and process used. The next three sections, namely Design and Development, Demonstration, and Evaluation, discuss the important implementation stages of the research. Finally in the section Discussion and Conclusions the results are summarized and potential future developments on the subject are outlined.

The sections that follow present auxiliary information such as References, Terms and Definitions and Search Terms. Appendices are given in the end.

2 Theoretical Background

This section describes the theoretical baseline for the rest of the thesis. First, the important definitions as used in the research are introduced and then the existing research in the fields of architectural patterns and cloud related to the subject of the thesis is reviewed.

2.1 Important definitions

2.1.1 Cloud computing

"***Cloud computing*** is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models" (NIST: Mell & Grance, 2011). In this model a vendor provides cloud computing services to consumers. In that sense consumers are those who rent cloud computing resources from a vendor. Consumers typically use these resources either for their own needs or for building their own systems which are in turn offered to their own clients or to end-users.

The five essential characteristics mentioned are ***on-demand self-service***, ***broad network access***, ***resource pooling***, ***rapid elasticity***, and ***measured service*** (NIST: Mell & Grance, 2011).

On-demand self-service allows cloud consumers to provision on their own the computational resources they need without redundant interaction with the vendor.

The capabilities provided by vendors are available over the network through a standard protocol and often represent coherent services – this is the broad network access. This setting is similar to the service-oriented model, where software services are available on the network; in that sense cloud computing extends the service-oriented model, but it also uses it. The broad network access characteristic means that using and integrating the provided capabilities over the network is an important aspect when architecting cloud systems thus featuring loose coupling similar to distributed systems.

As for resource pooling the computing resources are provided to multiple consumers through dynamic reallocation hidden from the consumers. The consumers are served in a dynamic multi-tenant model. Multi-tenancy means that different consumers may use the same physical or virtual hardware resources or instantiations of software systems at the same time with their data and resources still kept separated – the consumers are simultaneously tenants in the same resource (Bass, et al., 2012). Further, the resources are abstracted in a way that almost hides their concrete distribution and location; "almost" because sometimes they are provided with the notion of geographical areas. The consumer has "no control or knowledge over the exact location of the provided resources" (NIST:

Mell & Grance, 2011). Thus a system in the cloud needs to be designed from a location-agnostic distributed perspective.

Rapid elasticity is concerned with the ability to manually or even automatically scale up and down resources, i.e. allocate to the consumer more or less resources respectively. This characteristic influences the way systems are architected because as scaling might even happen automatically, systems should be designed in a way that utilizes this opportunity or at least does not suffer from it (Varia, 2011, p. 9).

Finally, the service is measured – the resources used are metered and usually there is a cost of usage that the consumers pay. As consumers can determine their resource usage and thus the costs they pay, this brings forward costs of operation as a non-functional requirement which needs to be addressed in the software architecture (Krishna & Jayakrishnan, 2013, pp. 89-90).

Other implications of these cloud characteristics upon architecture are increased importance of concurrent, asynchronous and parallel processing, data transfer and partitioning (Krishna & Jayakrishnan, 2013, p. 83).

Before going into the service models an important term needs to be introduced. According to NIST (2011), ***cloud infrastructure*** is "the collection of hardware and software that enables the five essential characteristics of cloud computing". It can be viewed in terms of two layers: physical layer consisting of the hardware resources and an abstraction layer consisting of "the software deployed across the physical layer, which manifests the essential cloud characteristics".

The three service models as illustrated in Figure 1 are ***Software as a Service (SaaS)***, ***Platform as a Service (PaaS)*** and ***Infrastructure as a Service (IaaS)*** (NIST: Mell & Grance, 2011). In the figure, as well as often in the literature, systems deployed in the cloud are denoted with the term "cloud applications". Those cloud applications utilizing the specific cloud characteristics and models are in turn called ***cloud-native applications*** (Fehling, et al., 2014a, p. 5).

In essence SaaS provides cloud computing consumers with the ability to consume a cloud-running service/application with a very limited possibilities for configuration (NIST: Mell & Grance, 2011, p. 2). As in this service model the role of the consumer is limited to only using resources, this model falls outside the scope of this research – cloud computing consumers do not design and build applications on their own in that service model.

PaaS gives the cloud computing consumer the ability to deploy to the cloud infrastructure and build their own applications (possibly providing them as SaaS later) on top of a mostly predefined infrastructure (NIST: Mell & Grance, 2011, p. 2). This means that architecting software solutions applies to this service model. Designing for this cloud service model is constrained by what the vendor offers, thus a number of technical architectural constraints are present and need to be accounted for depending on the chosen vendor.

Lastly, IaaS goes further allowing the cloud computing consumer to "provision processing, storage, networks, and other fundamental computing resources" (NIST: Mell & Grance, 2011, p. 2). This service model gives the most freedom to cloud computing consumers removing many of the limitations from PaaS. This

freedom brings software design and development for this service model closer to traditional environments allowing for creating more complex systems with more complex architectures.

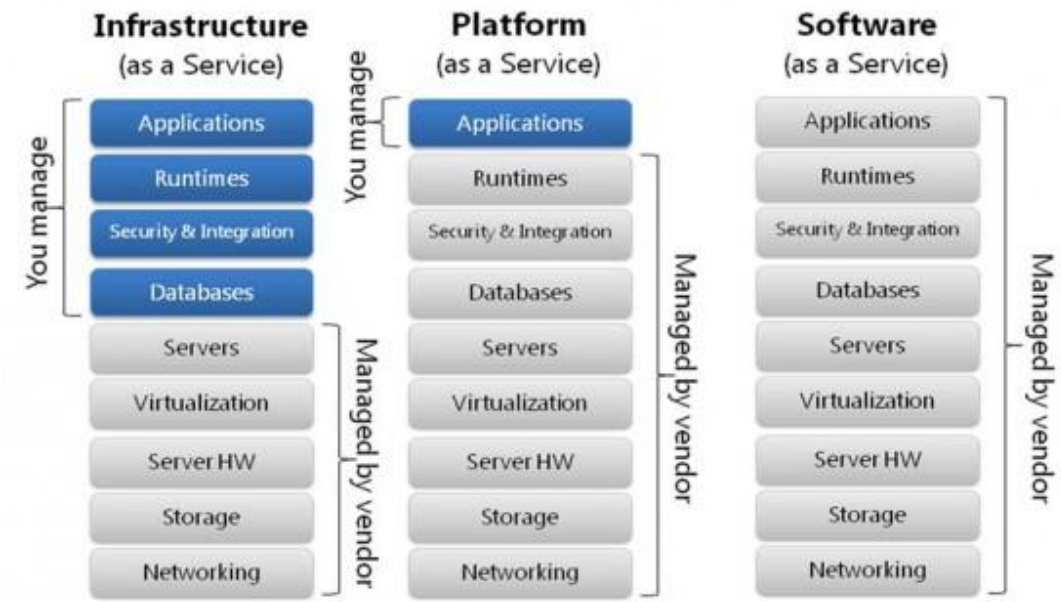

Figure 1 Cloud service models (CIO Research Center, 2010).

Finally, the four deployment models are private, community, public and hybrid cloud. The private cloud is operated for a single organization only, the community cloud operates for a community, the public cloud is available to anyone and the hybrid cloud is "a composition of two or more distinct cloud infrastructures" (NIST: Mell & Grance, 2011, p. 2). As the private cloud is intended to serve an organization, then this organization could use it for any purpose including building own systems which falls within the scope of the paper. As for a hybrid cloud, the existence of such a deployment model emphasizes the broad network access characteristic.

Cloud computing itself and the applications built on top of cloud infrastructure are inherently service-oriented (Guha, 2013) and distributed (Zhang, et al., 2010; Fehling, et al., 2014a). Computing resources are provided as services over the network with the three service models discussed above. Cloud applications use these services and due to the cloud computing characteristics, especially resource pooling and broad network access, the system architecture also needs to be more or less service-oriented and distributed. Thus loose coupling and integration of distributed services are clearly outlined as important aspects of architecting for the cloud.

2.1.2 Software architecture

Although numerous attempts have been made to provide a definition for ***software architecture***, "no universal definition of software architecture exists" (Clements, et al., 2010, p. 3). On the web site of the Software Engineering Institute (SEI) hundreds of definitions are gathered. Yet, as Clements et al. point out, although there is still no universal definition, all definitions gathered have a "centroid" which is relatively apparent and "from which very useful progress could be made". Two definitions that capture this centroid are the following:

- Software architecture is the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (ISO, 2011a)
- "Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled" (Woods, 2014)

The first definition is taken from the ISO/IEC/IEEE 42010:2011, *Systems and software engineering — Architecture description* standard and is more formal, whereas the second definition is closer to practice with a project focus. These show several important concepts important for understanding what software architecture is. Here some of the concepts crucial for this paper are outlined.

First, software architecture is a "conception of a system… [and it] may exist without ever being written down" (ISO, 2011a).

Second, in the ISO/IEC/IEEE 42010:2011 standard "every system is considered in the context of its environment" (ISO, 2011a). Environment is meant as a complex system of people, organization and system concerns with all their aspects. Then the software architecture of a system is observed and is influenced by the system environment. In the context of this thesis two broadly-defined types of system environments are investigated: the ***traditional*** and the ***cloud environment***. As already mentioned in the introduction section "traditional" is used as a synonym for "not cloud".

Finally, the second definition emphasizes the decision aspect of software architecture that may not be immediately obvious from the first definition. Designing a software system is a complex decision-making process. The ISO/IEC/IEEE 42010:2011 definition notes that the software architecture is embodied in its design principles. Rozanski & Woods (2011) describe an ***architectural design principle*** as "fundamental statement of belief, approach, or intent that guides the definition of your architecture". It may be based on, for example, personal experience, customer requirements or established good practices; examples of such principles can be found in (Vogel, et al., 2011, pp. 118-140). Apart from architectural principles, architectural decisions may also be triggered by, for instance, technological or business constraints (Clements, et al., 2010, p. 7) or by a lack of knowledge in a specific area. Regardless of the trigger, architectural decisions could be rather implicit and undocumented. Architectural decisions shape the software architecture and their significance has been widely accepted. Documenting them has also become a hot research topic in the last few years.

Clements et al. (2010, pp. 6-9) discuss another important issue: the difference between architecture and design. According to them "architecture is design, but not all design is architectural", thus differentiating between architectural and non-architectural design and design decisions. Discerning between what is architectural and what is not is a subjective matter that also depends on the context. This underlines the importance of the discussion of the centroid above, but as the terms are rather abstract, they are also subject to interpretation. Another case in which the difference between architectural and non-architectural design is important is patterns.

2.1.3 Patterns

A pattern "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, et al., 1977). Although this definition was initially intended for the building architecture field, it became widespread and pattern-related studies have been conducted in many fields such as organizational management, education and software.

In the software engineering context the consideration of patterns and their proper utilization has become an important part of effective and high-quality design. Patterns are often discussed in terms of the pattern as a description supplemented with a model of a solution for a given context and a ***pattern instance*** which is a concrete implementation of the pattern (Kobayashi & Saeki, 1999).

As outlined previously, the concepts of software design and software architecture are related. Following this logic, patterns can be viewed as architectural and non-architectural. The categorization of patterns presented by Buschmann, et al. (1996) confirms this view, defining three categories of patterns: ***architectural patterns***, ***design patterns*** and ***idioms***. As with the discussion about software architecture, the difference especially between architectural and design patterns is quite subjective and is sometimes hard to call. That is the reason why in some literature this is not even attempted (Fowler, et al., 2002). In other cases it is arguable, but in this thesis differentiation is fundamental.

An ***architectural pattern*** describes "a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them" (Buschmann, et al., 1996). Architectural patterns encompass a well-proven experience in building software systems by providing a generic solution with specific characteristics. As pattern-oriented software architecture research has shown architectural patterns can provide "a structural framework for a software system", but they do not represent a complete software architecture; they need to be further refined and specified to the system needs (Buschmann, et al., 1996). Attractive as patterns are, overcommitting to them is hazardous. As with any other type of pattern, architectural patterns need to be well-understood and used in their proper context. If so, they effectively help the architect in taking important design decisions with less risk.

According to the classic book of Gamma et al, a ***design pattern*** provides "a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context" (Buschmann, et al., 1996). The difference with architectural patterns lies in the impact on the system – architectural patterns define the fundamental structure to subsystems and other building blocks, whereas design patterns refine a specific subsystem design. Yet, as already stated, the boundary is not always obvious nor even possible to be defined. For example, in smaller systems such as a stand-alone mobile application, choosing a design pattern may have such a significant impact on the system architecture that the design pattern essentially becomes an architectural pattern.

Idioms describe "how to implement particular aspects of components or the relationships between them using the features of the given language" (Buschmann, et al., 1996). Thus idioms are language-level patterns. It is important to also note that design patterns are not dependent on a programming language, so the difference between design patterns and idioms becomes quite clear.

Architectural patterns are the focus of this research. Thus from now on they will be referred to simply as "patterns"; where another type of pattern is meant it will be explicitly stated.

Design patterns and architectural patterns are often cumulatively referred to as ***software patterns***. Apart from that in this paper ***cloud patterns*** is used to refer to patterns in the context of a cloud system environment whereas all other patterns are referred to as ***traditional patterns***. Note that both cloud patterns and traditional patterns should also be considered as software patterns. Further, when considering applicability for cloud, two types of cloud architectural patterns are differentiated: ***cloud infrastructural architectural patterns*** (CIAP) and ***cloud application architectural patterns*** (CAAP). CIAP deal with the specific software architectural concerns when building a cloud infrastructure, whereas CAAP are the architectural patterns suitable for building cloud-native applications; CAAP are the focus in the thesis.

2.1.4 Pattern evolution

Pattern evolution as treated in this thesis encompasses the changes that a pattern as a description or a template of a solution goes through. Such changes could be instantiated by a change in the context, the problems solved or any other external to the pattern factor which the pattern depends on. For example, when a new context (such as cloud) emerges a pattern may need to be adapted to be adequately fit for it. Similar to evolution in the Darwinian sense, the pattern as it was before the change does not cease to exist – the pattern continues to exist and is still applicable and important in its original context, but a variation evolves from that original pattern; this variation needs to be researched and documented. The extent to which such a variation differs from the original pattern determines whether the variation is treated as a pattern variant or as a completely new pattern. There are no strict rules to determine the boundary between calling a variation a

pattern variant and differentiating it as a new pattern, but as with biological evolution the differences need to be evident and significant in order to motivate the separation.

This discussion of pattern evolution also explains the categorization of patterns into transfer, evolution, innovation and oblivion cloud applicability categories. Those traditional patterns that can transfer without evolution to the cloud context are suitable for the cloud context without changes, i.e. they do not need to evolve. The group of evolution captures those patterns that have evolved in order to adapt to the new context. With respect to the discussion about pattern evolution above, these patterns could either just add a new variant to their current description, or a new pattern with a different name could be specified. The way in which understanding the connection of Homo sapiens to its ancestors in anthropology helps to understand better our biology, realizing the evolutional relation between patterns and the reasons for pattern evolution could help better understand them. The third group of patterns in the categorization is of those that originate from the cloud environment, i.e. they have emerged from it. The last group, the group of oblivion contains those patterns that have not adapted to the new environment and are not fit for it. Oblivion in that sense is somehow overstated, because, as stated previously, they do not seize to exist, but they should probably be considered less often in the cloud context. On the other hand such patterns could raise the questions "Is there a way to adapt these? Do they possess characteristics that could be of use?"

2.1.5 Frameworks, architectural styles, and architectural tactics

The concept of a framework is somewhat related to architectural patterns, because it is considered with reuse. "A framework is a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them" (Buschmann, et al., 1996). In that sense, patterns and frameworks both present a possibility for reuse and from a pattern perspective, as Buschmann et al. say, frameworks can be looked at as coarse-grained patterns "for complete software systems". On the other hand, frameworks can be built from pattern instantiations i.e. "patterns as building blocks". So patterns can be thought of as reuse in the abstract, whereas frameworks are reuse in the concrete.

Architectural style is another term closely related to architectural patterns. The two terms are even often used in the field as synonyms. As Clements, et al. (2010) note the difference is subtle, but it exists. Architectural styles are broader as they capture the general approach towards the architecture. Often this approach is put into context using architectural patterns. Further, patterns focus on the context, the problem and the solution, but styles are focused mainly on the solution i.e. representing an approach. Very often the architectural styles also have a same-named pattern counterparts; for example PIPES AND FILTERS as a term has the meaning of an architectural style or a pattern depending on the context of use. The difference between the two terms, although existing, does not have a very

practical implication (Rozanski & Woods, 2011), but needs to be shown in this report as the focus here is on patterns, rather than styles.

The last term discussed here with relation to architectural patterns is architectural tactics. "An architectural tactic is an established and proven approach you can use to help achieve a particular quality property" (Rozanski & Woods, 2011). Tactics are simpler than patterns, they are a general guideline on how to address one specific aspect of an architecture, say modifiability (Bachmann, et al., 2007) or availability (Scott & Kazman, 2009). Bass et al. (2012) present tactics as building blocks of architectural patterns – the patterns often comprise of tactics. They also show that tactics usage augments patterns by bringing new characteristics to patterns. An example of a study of the impact of tactics on patterns is presented by Harrison, et al (2010).

2.2 Related work

Architectural and design patterns have already been researched for almost two decades. The next subsections discuss the literature on subjects related to the thesis such as pattern languages, cloud application architectural patterns, the relations within the term tuples (architectural patterns, pattern languages) and (pattern evolution, pattern-oriented system development and evolution).

It is important to note that although some of the sources discussed in the next paragraphs are on design patterns, some of the observations made in them are also applicable to architectural patterns and represent important knowledge – this is the reason for their inclusion.

2.2.1 Patterns and pattern languages

As shown previously, there are several important terms in the patterns field. The boundary between some of them is sometimes quite hard to draw e.g. architectural and design patterns. Buschmann, et al. (2007b) discuss many common reasons for misinterpretation and misuse of the term "pattern" and "architectural pattern" in particular. Such misinterpretations have a significant impact on the literature about patterns. Avgeriou & Zdun (2005) state that the literature regarding patterns is "voluminous and heterogeneous", sometimes obeying to very different philosophical principles. In addition, sometimes specific artefacts, concepts or practices such as algorithms, data structures, technologies, even requirements are mistakenly stated to be patterns, but in fact they are not. To make things worse, pattern forms are sometimes intentionally used for documenting things other than patterns, although explicitly noted they do not represent a pattern e.g. (Fehling, et al., 2011) or in other cases the term "pattern" is just consciously treated loosely "to mean a recommended way to do things" e.g. (Dykstra, et al., 2014). Although such conscious or unconscious usage may have its benefits in terms of clarity of presentation, it makes it harder to find patterns in the full sense of the term. Furthermore, Hohpe hints at a practice which is sometimes abused for marketing and sales purposes in software engineering literature – including "a popular buzzword [such as "cloud"] and append[ing] the word "patterns" to suggest that

this particular title contains a substantial treatment of the subject matter and is organized as a collection of easy to digest chapters, which promise to provide solutions to recurring problems" (Fehling, et al., 2014a, p. vii).

On the other side of the coin, there are enough books and articles which represent a very valuable and trustworthy resource on patterns used in a traditional environment. In his foreword, Hohpe (Fehling, et al., 2014a, p. vii) further notes that most of these "good" resources are a result of "a true community effort" exemplified by the Pattern Languages of Programming conference being a birthplace for many quality books and articles. This puts emphasis on the significance of the community on creation and validation of quality materials. Hohpe writes further (Fehling, et al., 2014a, p. viii) that "most successful patterns books strike a delicate balance between academic rigor and real-world applications. The academic world brings depth of thinking and a clear structure, while the industry contributes the required validation and real-world examples." From that point of view, many books are inspired by practice and are useful for practitioners despite having some terminology shortcomings, but the balance remains key – after all patterns are also intended to provide a shared vocabulary for professionals (Buschmann, et al., 2007b) and discrepancies in terminology hinder the achievement of such a goal.

Up to here, patterns have been discussed in either a standalone, individual manner, or as taxonomies and classifications based on some guiding principle, for example the level of abstraction. Another examples of such classification are also some existing wiki systems such as the first ever wiki (Pattern community, 2013), the AWS Cloud Design Patterns (Community, 2013) and the wiki supporting Fehling, et al. (2014a) - (Fehling, et al., 2014c).

But patterns are often accumulated and represented in the form of a pattern language (Schmidt, et al., 2000; Kircher & Jain, 2004; Buschmann, et al., 2007a; Hohpe & Woolf, 2003; Fehling, et al., 2014a). ***Pattern language*** is "a network of interrelated patterns that define a process for resolving software development problems systematically" (Buschmann, et al., 2007b). Therefore, the difference between a taxonomy/classification and a pattern language is that the former does not define a systematic process that can be followed, it just groups the related patterns based on a criteria.

Alexander et al. (1977) in fact discuss patterns organized in pattern languages. This has inspired the creation of a large number of pattern languages over the years e.g. parallel programming (Ortega-Arjona, 2010), concurrency (Schmidt, et al., 2000), distributed computing (Buschmann, et al., 2007a). Avgeriou & Zdun (2005) note that although this is useful for treating a specific domain, it does not allow for integration of the overall architectural knowledge. This has ultimately led to their paper in which they propose an architectural pattern language which "acts as a superset of the existing architectural pattern collections and categorizations". They put emphasis on the relationships between the patterns, noting that patterns "have already been elaborately described before" and also link to the existing pattern languages. Even though the paper is rather popular in the researcher community

(about 170 times cited according to Google Scholar), there is still no commonly accepted classification of architectural patterns.

2.2.2 Cloud application architectural patterns

In terms of literature concerning cloud a significant number of articles, papers and books have been published. As stated by Marinescu (2013) cloud computing "is expected to shift from building the infrastructure, today's main front of competition among the vendors, to the application domain". This current state is confirmed by the observation that most papers found during this research focus on building cloud infrastructure (hardware and software) or explaining the fundamental cloud concepts to different audiences and justifying the usage of cloud. An example of such a book is *Cloud Computing: Concepts, Technology & Architecture* (Erl, et al., 2014). But as Marinescu (2013) predicted, recently more and more articles and books are being published that discuss system development and application architectures for cloud, which is the focus of the thesis.

A few initial pattern languages for designing cloud applications have already been documented (Fehling, et al., 2014a; Erl & Naserpour, 2015). Apart from that, books and articles treating patterns outside a pattern language have also been published (Moyer, 2011; Wilder, 2012; Ghag & Bandopadhyaya, 2013; Strauch, et al., 2013). The availability of such resources, though, is not sufficient. Some of the cloud pattern literature found so far suffers from the same inconsistencies and weaknesses observed previously about pattern literature in general or from the transitioning of the focus in the cloud field. *Cloud computing patterns* (Fehling, et al., 2014a) with its supporting website (Fehling, et al., 2014c) is a book that illustrates this observation. The material presented there is based on a large-scale rigorous research on cloud computing patterns that started in 2011 (Fehling, et al., 2011) and was validated by practitioners. The pattern language represents a good frame of reference for taking important architectural decisions, but some of the patterns presented there are cloud service models (e.g. IaaS, PaaS, SaaS), cloud infrastructure concepts/technologies (e.g. hypervisor, elastic load balancer), architectural principles (e.g. loose coupling), or architectural tactics (watchdog and transactions). Despite such inconsistencies, overall the book represents an invaluable frame of reference to guide architectural decisions as it combines patterns with, for example, technologies, which can provide a more holistic view (Heesch, et al., 2011). The real issue is the vague recognition of the different terms and denoting them all as "patterns". For the context of this thesis, despite there being a chapter dealing with Cloud Application Architecture Patterns (Fehling, et al., 2014b), it falls short in discussing architectural patterns as in the restricted sense presented earlier. This leaves a knowledge gap that this report intends to fill.

2.2.3 Architectural patterns and architectural decisions

As discussed earlier, the notion of design decisions is a cornerstone for understanding the software architecture field. Many books and papers have been written on the matter of how a system is designed. In the recent years, numerous authors have published articles and books on architectural decisions and especially

their documenting. Architectural patterns (and patterns in general) are "a means of documenting software architectures" (Buschmann, et al., 2007b) as they capture practical architectural knowledge gathered from experience. Thus architectural patterns could be used in documenting the design decisions (Harrison, et al., 2007) either explicitly or implicitly.

Explicit documentation of design decisions, which includes documenting the usage of patterns, is a broad topic that has been and continues to be researched in books and articles such as those by Harrison, et al. (2007), Zdun, et al. (2008), Clements, et al. (2010), Heesch, et al. (2012).

In the latter case the power of patterns as an implicit documentation about design decisions comes from the fact they are well-known and are usually well-described in some pattern form. This means that the sole usage of a pattern should in general speak a lot. As Harrison, et al. (2007) note, though, "architecture patterns don't relieve the architect of all responsibility for documenting decisions" because there is much more than just a pattern. For example, even though a pattern is used its properties could be augmented by usage of special tactics which may be hard to detect. Further, as some patterns have alternatives with more or less different characteristics, the reasons behind preferring one pattern to another are lost.

Nevertheless, as with designing for any other environment, design decisions need to be informed and conscious. As Fehling, et al. (2011) mention "a simple migration of an existing application running on a single machine to the cloud only results in minor benefits and may even reduce the application's availability, if architectural principles are not followed properly". When building cloud-native applications or moving a traditional system to the cloud, it will most probably be useful to know the reasons whether, why and how a traditional pattern is suitable for the cloud context. Thus the results of this paper could be used to support decision making.

2.2.4 Documenting patterns

When it comes to documenting patterns, numerous pattern forms have been used (Cunningham & Cunningham, Inc., 2005). A ***pattern form*** is a predefined structured way of presenting the information about patterns. Rozanski & Woods (2011, p. 162) identified five important pieces of information common for most forms. These are name, context, problem, solution and consequences. The name is important as it ensures a common language when speaking about patterns. Therefore it needs to be both meaningful and memorable. The context "sets the stage for the pattern, explains its motivation and rationale, and describes the situations in which the pattern may apply". The problem part describes the problem that the pattern solves and the conditions that need to be met in order to consider applying the described pattern. The solution presents most often a model of the elements of the pattern and their interrelations. Finally, the consequences show what the results from applying the pattern are in terms of both positive (benefits) and negative (costs, trade-offs) aspects. This pattern form is very similar to the Compact form (Pattern Community, 2011).

Another popular pattern form is the Alexandrian or Beck form (Pattern Community, 2007) which adds a Forces section to the five sections stated above. This additional section discusses what makes the problem to be solved difficult and discusses alternative solutions why they fall short in solving the problem (Hohpe & Woolf, 2003, pp. xli-xliv).

The discussion of the pattern forms presented in this section helps later in this report to choose a way to document the findings.

2.2.5 Pattern-oriented system design and evolution through pattern evolution

As discussed earlier, software architecture is concerned with the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (ISO, 2011a). Most parts of this definition were elaborated earlier; in this subsection the emphasis will fall on system design and system evolution and where patterns and pattern evolution fit in.

On pattern-oriented system design and system evolution

When doing software architecture there are particular ways that will work better than others. This is the essence of "good" architecture (Vogel, et al., 2011, p. 118; Bass, et al., 2012) – what can be suitable for a web application may be highly problematic for an embedded system for example. The key is informed and reasonable design decisions need to be taken – architects must account for the characteristics of the environment, the requirements towards the system and many other factors. Some of these problems in front of the architect could be tackled through well-known and well-described patterns that suggest proven solutions. With that in mind, pattern-oriented software architecture has been one of the big trends in software architecture literature for more than a decade (Buschmann, et al., 1996; Schmidt, et al., 2000; Kircher & Jain, 2004; Buschmann, et al., 2007a; Buschmann, et al., 2007b). Pattern-oriented software architecture most often treats the design of a system as a composition and adaptation of architectural patterns. Architectural patterns address specific problems but often there are requirements which cannot be fully satisfied due to technological limitations, laws, business rules and so on. The result are trade-offs which need to be made but the available alternatives need to be carefully considered. There are methods for supporting architects in these decisions such as the Architecture Trade-off Analysis Method (SEI, 2014), but the decisions are for the architect to make – it is essential to know well the architectural patterns available, their characteristics and what properties they enhance in the designed system, and finally the pattern alternatives in order to find the most appropriate solution.

Going further, patterns are applicable not only during system design, but also during the later stages of the system development lifecycle. For example, Côté, et al. (2007) present a pattern-based model that is applicable during system design, system development and system evolution.

Looking at system development, Kobayashi & Saeki (1999) argue that the software development process can be looked at as a process of artefact evolution based on operations over pattern structures (a diagram documenting the pattern) and pattern instances. For the sake of simplicity this type of evolution will be called pattern instance evolution throughout the paper. Kobayashi & Saeki (1999) argue that examining and documenting that pattern instance evolution could provide for a well-engineered system.

As for system evolution, it in general obeys to several laws called Lehman laws (Lehman, et al., 1997). For example, systems need to be continuously adapted and to continuously grow otherwise they become less satisfactory for the end users. As this shows, systems need to evolve over time. This need has to be accounted for during the design phase. Many authors consider the usage of patterns itself as a way to design for change (Ram & Rajasree, 2003; Dong, et al., 2007; Zhao, et al., 2007; Dong, et al., 2010). Continuing to the actual system evolution, when the system has to change, there are overall two pattern-oriented ways of achieving this – pattern instance evolution and pattern substitution i.e. substituting a more appropriate pattern for an existing pattern. Ram & Rajasree (2003) argue that pattern instance evolution and pattern substitution are viable methods for implementing system changes. Further, pattern-oriented system evolution has been looked at extensively from a modelling perspective, most often in the form of formal model transformations (Dong, et al., 2010; Kim, 2013; Dong, et al., 2006). Other views and proposals on pattern-oriented system evolution include graph transformations (Zhao, et al., 2007) and even automated program transformations (Tokuda & Batory, 1995).

The role of pattern evolution

First of all, it is important to emphasize the touching points between pattern evolution as it is treated in this thesis and pattern instance evolution. The difference between these two terms is in the area of effect – pattern evolution is evolution of the description of the pattern in general and this has global effect, whereas pattern instance evolution is evolution of the concrete application of a pattern i.e. local effect. Also, pattern evolution could influence pattern instance evolution as the changes in the generic pattern description may be applicable to the pattern instance and make the result of applying the pattern better.

In the context of going towards cloud, system design is considered in terms of building cloud-native applications from scratch whereas system evolution could mean the natural process of evolution of a cloud-native application or transferring a traditional system to become cloud native i.e. to use the cloud characteristics. The latter case can be somewhat confusing as even though a system may be already hosted in cloud it may still remain a traditional system which does not utilize the cloud resources.

If building from scratch the architect picks the patterns that answer best the requirements and decides on trade-offs as discussed earlier. The knowledge of architects needs to be updated considering building for cloud is a relatively new topic and not much literature on it exists. Pattern evolution models exactly that transfer of knowledge.

The case when a cloud-native application needs to evolve is similar to building from scratch in terms of needed architectural pattern knowledge – the desired system properties and the possible alternative patterns, tactics and principles need to be known. The difference is that this process is in fact system evolution, i.e. the system exists and the context during system evolution is different from the context during system design. Again, knowing which patterns are suitable for cloud and how may help in this process.

Finally, when it comes to transitioning a traditional system to cloud the notion of pattern instance evolution comes handy. In fact, in order to execute the transitioning significant changes should usually be done, otherwise only limited benefits may be realised or even application availability could be influenced negatively (Fehling, et al., 2011). Each pattern instance should be rethought in terms of applicability for cloud; Strauch, et al. (2013) show how this could be done for the data layer. Although the pattern instance probably has some differences from the original pattern description, looking at pattern evolution could give the answers to the question "What changes need to be done?" Thus the information coming from knowing the pattern evolution could be very useful to architects which means that pattern evolution helps in the pattern-oriented system evolution process.

3 Research Method

This section discusses the choice of a research method for the thesis, the theoretical base of the method and how it was applied in the research.

3.1 Choice of method

The thesis was inspired by natural curiosity and initiated with the goal to model the development of the architectural patterns field in order to map the author's existing knowledge to the cloud context. Thus this paper is highly-oriented towards maintaining close relation to practice and providing tangible results centred on utility; utility is the focus of design science (Hevner, et al., 2004).

Most of traditional research methods discussed by Williamson (2002) were found inappropriate for the purpose of the work. The only method that was considered a candidate was systems development. It was observed that both systems development papers and design science papers often reference the paper by Nunamaker et. al. (1990-1991) in such a way that systems development looks either very similar or the same as design science. One of the differences between these two methods can be found in the different accents put when discussing them – systems development is more focused on building a system (although other types of artefacts are acknowledged), whereas design science discusses equally different types of artefacts including a system. This difference is evident in the discussion of systems development in information systems research (Williamson, 2002, pp. 147-155). In the same book the system developed is said to be used to prove a theory, rather than being itself an important result of the research as is the view in design science. Further, systems development is viewed as a part of a multi-disciplinary approach rather than a self-contained and self-sufficient method. After all that said and despite the similarity, design science is the method option that best serves the aims of the thesis.

3.2 Design science research

"Design science research is a research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artifacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem" (Hevner & Chatterjee, 2010a).

As shown on Figure 2 design science research is in constant interaction with the knowledge base and the environment to which it is applicable. The internal design cycle is a process of improvement of the research itself and the artefacts it produces. Based on these three cycles, the goals of a design science research are (1) to provide a solution to a problem and (2) to contribute to the knowledge base (KB).

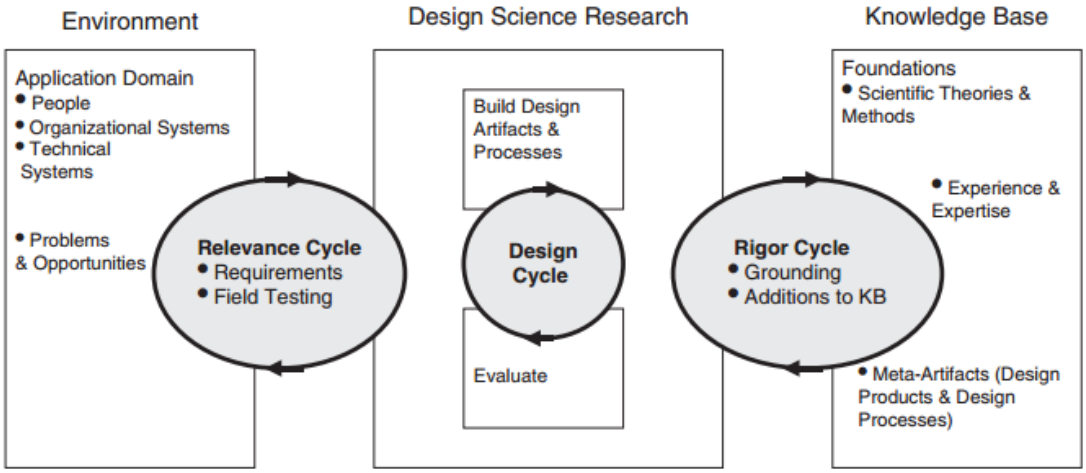

Figure 2 Design science research cycles (Hevner & Chatterjee, 2010b).

The concept of artefact is fundamental for design science – artefacts are the end goal of the research (Hevner & Chatterjee, 2010a). Hevner et al. (2004) broadly define the possible classes of IT artefacts as "constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems)"; Hevner & Chatterjee (2010a) add better design theories to this list. Instantiations are the most popular artefact produced in design science research (Arnott & Pervan, 2008), so dominant that it is easy to think of them as the only possible artefact. Contrary to this observation, this research is focused on developing a representation of the knowledge (the primary artefact), introducing the needed new vocabulary, and detailing the research process. These artefacts are discussed in more detail in the next paragraphs and are central for the rest of the paper.

The Design Science Research Methodology (DSRM) described in (Peffers, et al., 2007-8) (see Figure 3) and the guidelines from Hevner, et al. (2004) (see Table 1) provide the methodological basis for the research presented in this paper.

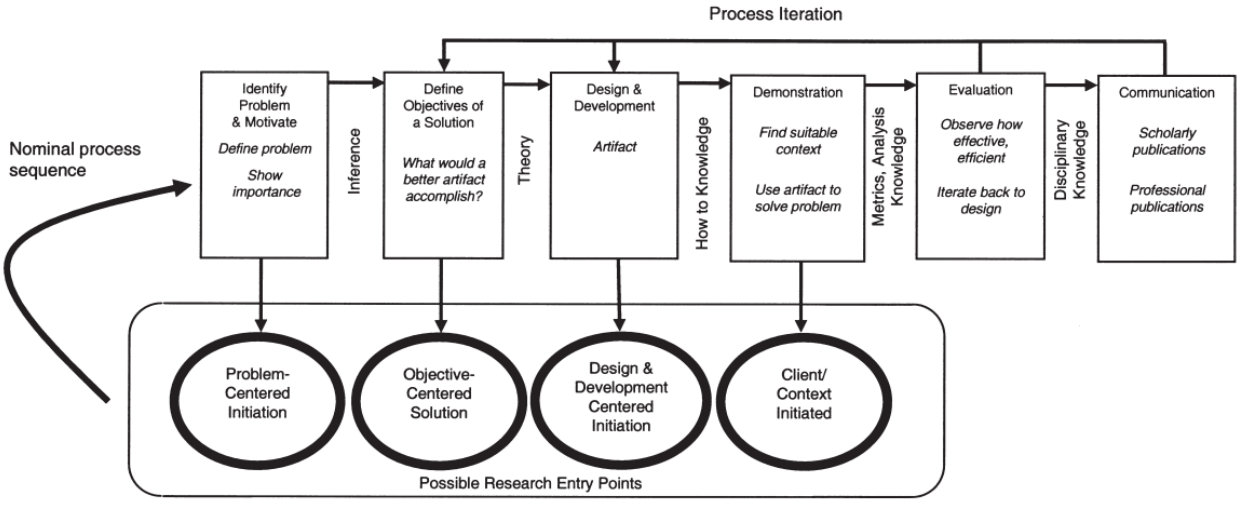

Figure 3 DSRM Process Model (Peffers, et al., 2007-8).

Guideline	Description
Guideline 1: Design as an Artifact	Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation
Guideline 2: Problem relevance	The objective of design science research is to develop technology-based solutions to important and relevant business problems
Guideline 3: Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods
Guideline 4: Research contributions	Effective design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
Guideline 5: Research rigor	Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
Guideline 6: Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
Guideline 7: Communication of research	Design science research must be presented effectively to both technology-oriented and management-oriented audiences

Table 1 Design Science Research Guidelines (Hevner, et al., 2004)

Although the guidelines presented in Table 1 do not have well-defined constructs and are subject to interpretation (Arnott & Pervan, 2008), they provide a structured path and direct the attention at important matters (Hevner & Chatterjee, 2010b). As Arnott & Pervan discovered, there are significant issues in the design science research so far with relation to these guidelines especially concerning evaluation, appropriateness of the research method and rigour in the process. This thesis work strives to recognize these weaknesses by either addressing them or by providing a discussion or a plan for additional research to cover these weaknesses.

The DSRM process on the other hand is consistent with these guidelines (Peffers, et al., 2007-8). It is both "a nominal process for conducting DS research" and "a mental model for the presentation of research outcomes". The 6 activities presented as rectangles in Figure 3 are based on ideas from previous research including the discussed guidelines. It is important to note that the process itself can be started from four possible entry points depending on the origins of the idea or problem and the availability of previous similar artefacts. The other important aspect is the iterative nature of the process. The communication and evaluation activities provide feedback which iteratively is incorporated in the process increasing its focus and in the artefacts increasing their utility.

3.3 Implementation

The instance of DSRM applied in this paper is shown on Figure 4. The research presented in this paper is positioned as a search for an objective-centred solution because it is "triggered by […] [a] research need that can be addressed by developing an artefact" (Peffers, et al., 2007-8) – it starts from the second possible entry point, objective-centred solution, which is marked in **bold** in Figure 4. The activities are retained as in Peffers, et al. (2007-8) and are also marked in **bold**, whereas the result of the activity or the particular actions taken during these activities in the implementation of the process for this research are listed in the same box below the activity heading and without special formatting.

The rest of this section deals with the different activities.

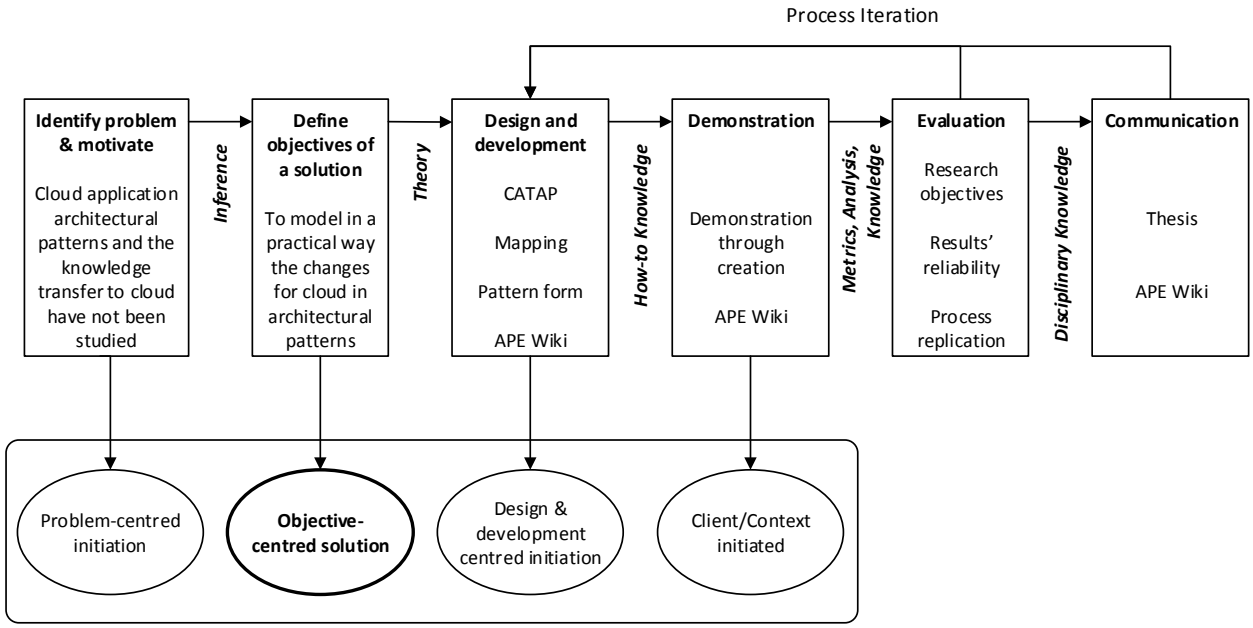

Figure 4 The thesis research methodology based on Peffers, et al. (2007-8).

3.3.1 Problem identification and motivation

As the overview of the literature showed much attention from both researchers and the pattern community has been paid to patterns overall and the way they are presented, modelled, and used in practice. Meanwhile, the review of related work on the subject established that there are hardly any (if any at all) resources on investigating the transitioning of architectural knowledge depending on context change such as the transfer to cloud. Moreover, cloud application architectural patterns have barely been studied before. Therefore the focus of the thesis is on that knowledge gap, namely the changes in the architectural patterns field related to designing and re-designing applications for the cloud.

This puts this paper as a positional or a transitional research for the field of patterns. The view on the subject presented in the thesis is that bridging the traditional to the cloud world is important for an effective transfer of architectural

knowledge and gaining better understanding of this process. The relations between traditional and cloud patterns are and will be relevant because the traditional and the cloud environments will continue to co-exist. Further, by looking at the development of patterns, important information about them is captured that is otherwise lost e.g. how patterns as definitions evolve due to context changes and what the rationale behind this change is.

This approach is similar to that of transformative learning (Mezirow, 1997) although the perspective here is not one of values and culture but one of architectural knowledge. The idea is that "to become meaningful, learning requires that new information be incorporated by the learner into an already well-developed symbolic frame of reference" (Mezirow, 1997, p. 10). One of the methods to effectively achieve this is concept mapping, but as Mezirow emphasizes critical thinking is still required. Concept mapping is exactly the direction of this research and the objectives for the solution are motivated by it; defining these is the second activity in the DSRM process.

3.3.2 Definition of solution objectives

As noted by Peffers et al. (2007-8) the objectives can be defined quantitatively or qualitatively. Due to the nature of the work the definition here is qualitative. The objectives are:

RObj1. to pay attention to the reasons why a pattern has transferred, transformed, emerged or became less relevant with respect to the characteristics of the patterns and their suitability for the cloud due to these characteristics;

RObj2. to distinctively present and outline these changes, so that it is clear what the differences are;

RObj3. to allow preservation and inclusion in the pattern description of the information about patterns that has been developed throughout the years and is still relevant;

RObj4. to keep the solution close to practice.

Providing an as detailed as possible description of the research process so that it can be replicated and expanded lies also in the main focus, although it is not an objective for the research itself.

While operationalizing these research objectives three research questions were formulated:

RQ1. How does the existing knowledge on architectural patterns relate to the cloud computing environment?

RQ2. Which characteristics of architectural patterns make them suitable for the cloud environment?

RQ3. How can architectural pattern evolution be documented effectively for usage in the practice?

So far in the previous sections a number of different types of resources were discussed such as research papers and articles, books, practitioner community-driven discussions; cloud platform documentation, analysis of usage in real-world contexts (industrial and scientific case studies) are added to this list in the next sections. Due to the pointed lack of research and the novelty of the field the dependence on non-scientific sources is needed. But considering the practical orientation of the thesis this is also a requirement. Community sources usually reflect better the practitioners' usage and concerns. This is strengthened by the fact that the pattern community is very active and a lot of the advances in the field are in fact driven by it (Buschmann, et al., 2007b). All these resources represent the "Theory" as shown on Figure 4, which connects the objectives defined above to the design and development activity.

3.3.3 Design and development

The next activity in the research process, design and development, is informed by the knowledge gathered previously and follows the directions outlined during the previous research activities.

Each of the research questions of the study motivates the creation of an artefact as shown on Figure 5. Artefact 4 is a special case that does not come with these research questions, but is relevant for the research – the reasons will be presented shortly. All these four artefacts comprise the primary compound artefact of the thesis - a representation of the changes in the architectural knowledge with respect to the introduction of the cloud context. Each of the artefacts provides an important perspective on the matter – designing and developing them is the subject of the design and development research activity. The relations between the questions, the artefacts and the end goal of the research are discussed in the following paragraphs.

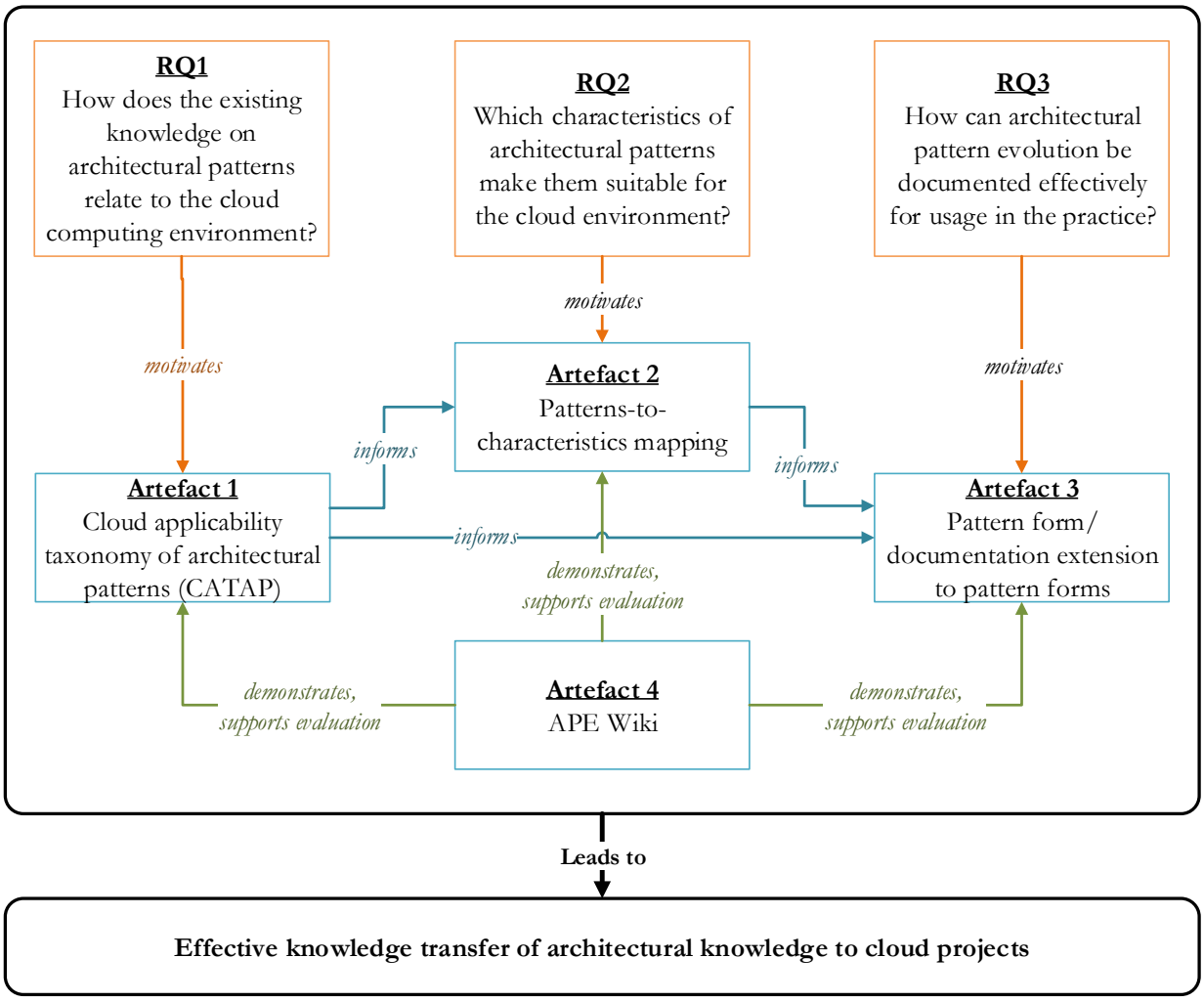

Figure 5 Relations between research questions, artefacts and goal.

Question 1 directs the investigation towards exploring the relevance of architectural patterns knowledge in the cloud. The relevance as a grouping factor determine the creation of four cloud applicability categories of patterns. These are:

- Transfer – the architectural patterns that move to the cloud without changes;
- Evolution – those patterns which are adapted to support cloud usage;
- Innovation – completely new patterns that have emerged for the cloud specifically;
- Oblivion – the category of architectural patterns that are not that relevant/suitable for cloud usage.

This idea serves as the basis of Artefact 1 – a cloud applicability taxonomy of architectural patterns, abbreviated and referred to as the CATAP in the paper. Although such a classification may be treated as a pattern language, the term "taxonomy" is preferred here, because the focus is not on solving a development problem systematically, but rather present the knowledge transfer.

Question 2 deals with the characteristics of patterns and their suitability for the cloud due to these characteristics. These characteristics are investigated based on Artefact 1. Investigating and documenting these characteristics provides insights into the development of patterns. For example, if a pattern is part of the Evolution group, then it has at least one "traditional" variant and a "cloud" variant. The "cloud" variant differs from the "traditional" in order to answer a demand from the cloud environment. The example can be expanded assuming that a new characteristic is brought to the traditional pattern to form the cloud variant. The observation of which characteristic has been introduced first could help practitioners change their perspective more easily by realizing and focusing on this change. This characteristic should be related to the environment so a list of such characteristics (Artefact 2) could potentially provide direction for thinking about how a pattern, which falls into the category of oblivion, for example, can be changed so that it gains relevance again. A more detailed discussion of these issues is provided in the respective subsection in the Design and development section.

Question 3 addresses the issue that the information about how patterns evolve has rarely been incorporated in pattern documentation. As discussed in the theoretical background, many pattern forms have been developed and patterns are actively documented, but the issue above has not been addressed. As previously said, this presents a hole in knowledge, a missing connection that if present could potentially help better understand patterns. All this motivates the design of a representational template (Artefact 3) that will solve this problem. A goal for it is to be possible for this template to be incorporated in the existing pattern forms, so that the existing documentations can be just extended, not completely rewritten.

Artefact 4, APE Wiki (Architectural Patterns' Evolution Wiki), is not inspired by an individual question, but is extremely important. First of all, wiki pages as a tool for collaboration were invented by the pattern community to facilitate the discussion of the topic (Buschmann, et al., 2007b). As one of the objectives for the research is to keep the closeness to practitioners, a wiki is a reasonable choice. Going further, the information architecture of the wiki itself also moves forward the complete solution advancing towards the ultimate goal. Although the other artefacts have relations between each other, they are not integrated and do not fully achieve the objectives of the research; the wiki fills that gap. Finally, the wiki connects well the end goal of design science to achieve utility through artefact creation, the objective of this research to be close to practice and the established way the pattern community documents, discusses and advances the field.

The information in the APE Wiki captures and represents the "How-to knowledge" link on Figure 5, leading to the next research activity: Demonstration.

3.3.4 Demonstration

The APE Wiki is relevant not only to the Design and Development activity, but also to the Demonstration activity. The other three artefacts are hard to be demonstrated because although they are relatively atomic artefacts, they do not represent the full solution. The wiki integrates them and provides the medium for demonstration. So the demonstration activity, although carried through the wiki, is an activity that is in fact performed for all four artefacts.

The demonstration of the artefacts is then analysed and evaluated in the next activity: Evaluation.

3.3.5 Evaluation

The Evaluation activity is again applicable for all artefacts. It is extremely significant as it is considered with "how well the artifact supports a solution to the problem" (Peffers, et al., 2007-8). The Evaluation also ensures the iterative nature of the research process.

The reliability of the results is ensured by rigour in the research, usage of multiple knowledge sources, and iterative re-evaluation of the models.

The evaluation method used is a triangulation of observational and descriptive evaluation methods (Hevner, et al., 2004). The APE Wiki acts as a case study or even a field study (observational methods), because it allows for collecting comments from practitioners who use the research results and can thus contribute to the evaluation. In terms of descriptive evaluation, both discussion of scenarios and informed arguments using information from the knowledge base are used. It is important to note that critical thinking is an inseparable part of this evaluation process.

Beyond the intended iterative nature of the research process itself, the collaborative nature of the wiki would allow the whole knowledge base created by this research to continue to evolve.

3.3.6 Communication

The final activity, Communication, is concerned with communicating "the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences such as practicing professionals, when appropriate" (Peffers, et al., 2007-8). In that sense, the two important audiences for this research are researchers in the field and pattern practitioners; respectively, this thesis report and the APE Wiki are the concrete media. This reflects Guideline 7 (see Table 1) from Hevner et al. (2004), recognizing that different audiences need different ways of communicating the results to. As the APE Wiki does not cover the details of the research process itself, the suggestion by Zmud (Hevner, et al., 2004, p. 90) to construct appendices is chosen as a supporting tool to convey this information to practitioners in a concise manner.

4 Design and Development

As described in the previous section most of the artefacts are developed in the design and development stage, driven by one or more of the research questions at hand. The consequent sections will look at them one by one, moving naturally from one to the other at the same time expanding the results and moving closer to fulfilling the goals.

4.1 Cloud applicability taxonomy of architectural patterns (CATAP)

The research starts with research question 1: RQ1 – **Error! Reference source not ound.** In order to answer this question, as discussed earlier in the section Research Method, the existing knowledge of architectural patterns should be explored in terms of its relevance for cloud, resulting in Artefact 1, the CATAP.

The subsections below describe, first, the process of constructing the CATAP, then illustrate the findings through a few examples and, finally, this section concludes with analysis of the results.

4.1.1 CATAP construction process

The perspective of investigating the relevance of the architectural patterns for cloud gives reason to constructing the CATAP with the following 4 categories of architectural patterns:

- Transfer – the architectural patterns that move to the cloud without changes;
- Evolution – those patterns which are adapted to support cloud usage;
- Innovation – completely new patterns that have emerged for the cloud specifically;
- Oblivion – the category of architectural patterns that are not that relevant/suitable for cloud usage.

Therefore, a stepping stone is to first find traditional architectural patterns. The aim of this research, though, is not to list all patterns ever discovered, but to provide a model for describing how the knowledge about these architectural patterns evolves with the change of the system environment, thus creating a taxonomy rather than a pattern language.

In order to construct the CATAP the following 4-stage process is used:

1. Look through existing scientific sources about traditional patterns and extract the architectural patterns;
2. Search in literature for the patterns found during the first stage about their use in cloud;
3. Search for new cloud-inspired patterns;
4. Construct the cloud applicability taxonomy of architectural patterns.

The following subsections provide more details on these stages.

4.1.2 Selection of architectural patterns for the CATAP

The first 3 stages of the process presented above are all about discovering the architectural patterns and recognizing whether they are new or have history which needs to be tracked. These 3 stages are presented below.

Extracting traditional architectural patterns from literature

Literature on pattern taxonomies shows that numerous taxonomies (such as the one based on level of design presented earlier) have been proposed. Many authors have also concentrated on establishing pattern languages for concrete domains or classes of tasks e.g. parallel programming (Ortega-Arjona, 2010), concurrency (Schmidt, et al., 2000), distributed computing (Buschmann, et al., 2007a). Apart from that, oftentimes no attempt for distinguishing design and architectural patterns is made (Fowler, et al., 2002), or one and the same pattern is named differently across different sources. In this variety of perspectives, representations and classifications it is not a trivial task to find a common basis.

A viable approach would be to search in these and other books and resources on software patterns and extract the architectural patterns from there. Following that line of thought, the books cited above served as a starting point of the research. The aim at this initial literature review was to establish a manageable and at the same time big enough to be representative set of architectural patterns which all comply with the definitions expressed in the paper. This set later on would be used as a baseline for comparison with already existing explicit classifications of architectural patterns. This is the high-level process behind this stage of the CATAP construction process.

The criteria used to filter out architectural patterns from other types of patterns and pattern-like constructs comprises of the following 3 factors applied one after the other acting as a sieve:

- The construct under consideration must be a pattern in Alexandrian sense (Alexander, et al., 1977) – this factor filters out non-patterns. A pattern in the Alexandrian sense should describe:
    - a problem which is broad enough, so that it can be reused in different cases, and at the same time is not too abstract so that it does not become unspecific and does not loose connection to its context;
    - a solution to that problem which provides guidance but not exact implementation, so that one "can use this solution a million times over, without ever doing it the same way twice" (Alexander, et al., 1977)
- The problem being solved by the pattern must be a software problem. Although this factor is rather clear because of the context of the work, it is listed here for completeness.

- The pattern must have fundamental architectural significance; this factor filters out design patterns and idioms, leaving architectural patterns. A rule of thumb for assessing a pattern for this factor is to ask questions about the impact of the pattern if it is introduced, changed or removed. If the pattern significantly affects the communication flow between subsystems/ components or impacts the subsystem/component decomposition, then this is most probably an architectural pattern.

Studies extending this one could either use that same criteria to filter through other architectural pattern languages and classifications or to extract patterns from pattern languages and sources which do not explicitly recognize the patterns they talk about as architectural.

Following the research model for this stage explained above, initially a set of 10 architectural patterns was extracted from the classical books on patterns - *Patterns of Enterprise Application Architecture* (Fowler, et al., 2002), *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Hohpe & Woolf, 2003), the *Pattern-Oriented Software Architecture* series (Buschmann, et al., 1996; Schmidt, et al., 2000; Kircher & Jain, 2004; Buschmann, et al., 2007a; Buschmann, et al., 2007b) – and was triangulated with other literature. The architectural patterns comprising that set were noted, keeping reference to their original source and the category they were assigned to in that source. Based on the criteria above representing the view of architectural patterns expressed in this paper and the set of 10 architectural patterns extracted from literature, a number of architectural pattern classifications were evaluated. In result, the research made by Avgeriou & Zdun (2005) proved to be the closest in the perspective on what architectural patterns are and at the same time it contained all 10 architectural patterns extracted during the initial base study. This gives enough ground to use the results of Avgeriou & Zdun's (2005) extensive study as a baseline for this research.

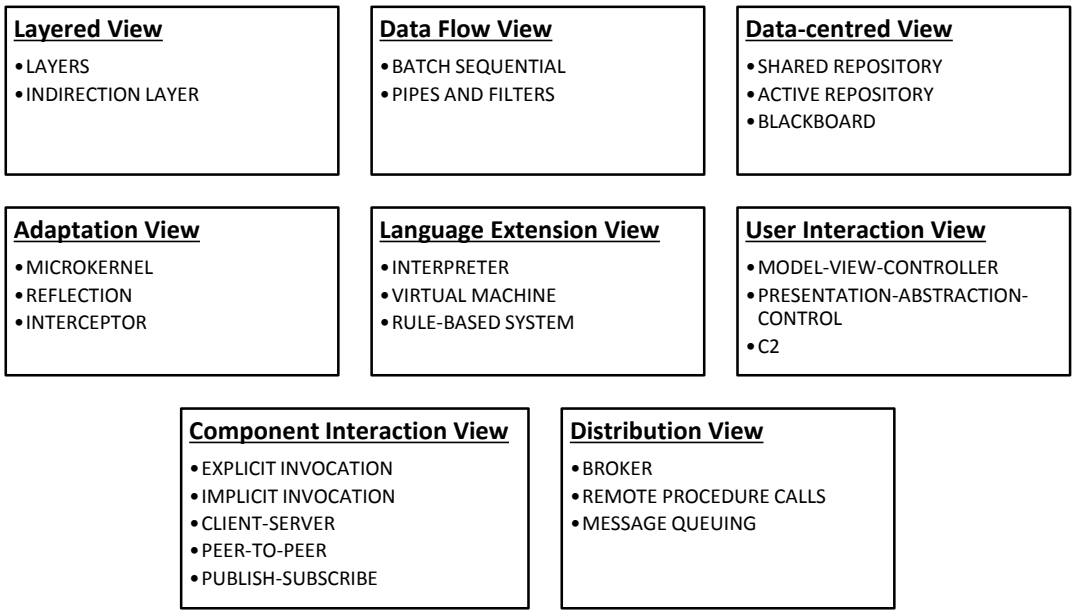

Figure 6 Traditional architectural patterns – a pattern language from (Avgeriou & Zdun, 2005).

Figure 6 lists the patterns found in their study. They are organized into 8 ***architectural views*** – the reference to these architectural views is kept, because it brings context information with it. An architectural view is "a representation of a system from the perspective of a related set of concerns" (Avgeriou & Zdun, 2005, p. 4). The views Avgeriou & Zdun use to organize their pattern language are as follows:

- The Layered View looks into a system in terms of how it can be structured as interacting parts.
- The Data Flow View looks into how a system's components can sequentially process and transform data streams.
- The Data-centred View is concerned with utilizing a central repository of data accessible by multiple components.
- The Adaptation View looks at systems as consisting of invariable and adaptable components.
- The Language Extension View looks at the system as a part of the external world which it communicates with through a form of abstraction.
- The User Interaction View is concerned with "the runtime structure of components that offer a user interface" (Avgeriou & Zdun, 2005).
- The Component Interaction View focuses on communication through messages which also preserves the components' autonomy.
- The Distribution View looks into systems as distributed in a networked environment.

Differentiating the patterns in such views is important because software architecture must take into account different aspects of the produced system – distribution, processes, communication, to name a few (Kruchten, 1995). Each pattern usually addresses one or more of these views, based on the perspective given from the formulated problem in the pattern description. So in the different views, different questions are posed in front of the architect, which means completely different solutions may be found depending on the view – the resulting system architecture is the result of taking all these into account, evaluating them and choosing and modifying a set of them which answers most of the questions asked – most, because trade-offs are inevitable (Kazman, et al., 2000). Therefore, it is important to look also at potential combinations of patterns and relations between these patterns later in this section.

Finding documented usage of traditional architectural patterns in cloud environment

As shown in the report so far, literature on traditional architectural patterns is abundant; literature discussing at least some of the traditional architectural patterns in a cloud context is also present. The focus, though, most often currently falls onto the most popular architectural patterns such as LAYERS, PIPES AND FILTERS, MESSAGE QUEUING. For the rest, literature can help up to the point to provide a foundation for analysis of the applicability.

The primary criteria when evaluating an architectural pattern for its applicability for cloud is how well it supports the five essential cloud characteristics of on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service (NIST: Mell & Grance, 2011). In this analysis an important role take also non-scientific sources such as blogs of companies providing cloud services and systems, community wikis.

Such an evaluation can only be considered as a starting point for future work, though, as it is not based on practice and real-world cases which should in general be the source of insights for patterns. The process at this stage is very similar to the one from the first years of interest in software patterns as the field of cloud system architecture is still to be studied and developed.

Search for new cloud-inspired architectural patterns

So far there has been very little scientific discussion of architectural patterns for building applications for cloud. Most of the literature on cloud patterns is centred either on CIAP or on foundational principles of building cloud applications and primarily on design patterns. Similar to the literature about traditional patterns, little effort has been made to discern architectural from design patterns and even patterns from tactics or cloud platform specific technologies and techniques, for example. Therefore, although scientific literature can also be used, the biggest sources for extracting architectural patterns discovered because of cloud can be found in sources such as community wikis, technical documentation, blogs of companies building cloud products and practitioners' blogs and articles. Examples of such resources are a wiki for Amazon Web Services patterns (Community, 2013), the High Scalability community blog (Community, 2015), architecture guidance for Microsoft Azure (Microsoft Corporation, 2014), Netflix's technical blog (Netflix, 2015). In addition, there currently are few attempts to invent patterns during professional research (Bykov, et al., 2011) which only strengthens the observation that action is required to address the need for building systems for cloud.

4.1.3 The CATAP

The final stage of constructing CATAP is to look at the architectural patterns discovered in the previous stages and to group them in the 4 cloud applicability categories – transfer, evolution, innovation and oblivion. This is in fact the process of constructing Artefact 1, the CATAP. It is important to emphasize that the CATAP does not represent a pattern language i.e. it does not offer a systemized way to solve problems – it is rather a taxonomy which enables a simplified model of thinking when considering the knowledge transfer to cloud.

Figure 7 represents the CATAP. All architectural patterns except the one in the Innovation category are part of the Avgeriou & Zdun (2005) classification presented before.

Transfer

- LAYERS (Layered view)
- INDIRECTION LAYER (Layered view)
- BATCH SEQUENTIAL (Data-flow view)
- PIPES AND FILTERS (Data-flow view)
- SHARED REPOSITORY (Data-centred view)
- ACTIVE REPOSITORY (Data-centred view)
- BLACKBOARD (Data-centred view)
- INTERCEPTOR (Adaptation view)
- MODEL-VIEW-CONTROLLER (User Interaction view)
- PRESENTATION-ABSTRACTION-CONTROL (User Interaction view)
- C2 (User Interaction view)
- IMPLICIT INVOCATION (Component Interaction view)
- PUBLISH-SUBSCRIBE (Component Interaction view)
- MESSAGE QUEUING (Distribution view)

Evolution

- -

Innovation

- COMMAND QUERY RESPONSIBILITY SEGREGATION

Oblivion

- MICROKERNEL (Adaptation view)
- REFLECTION (Adaptation view)
- EXPLICIT INVOCATION (Component Interaction view)
- CLIENT-SERVER (Component Interaction view)
- PEER-TO-PEER (Component Interaction view)
- BROKER (Distribution view)
- REMOTE PROCEDURE CALLS (Distribution view)

Figure 7 Cloud applicability taxonomy of architectural patterns (CATAP).

The Language Extension view was omitted from the discussion as the way it is presented in (Avgeriou & Zdun, 2005) supposes an approach a bit closer to the hardware and operating system environment, which often is not the case for cloud applications. Despite that, at least some of the patterns in this view e.g. RULE-BASED SYSTEM, if looked from the perspective of their ideas can be translated to scenarios applicable for cloud. This, though, falls out of the scope for this paper.

4.1.4 Discussion of patterns with respect to the CATAP

The following paragraphs discuss in depth some of the patterns above and why they were classified in their respective groups. More details and a short summary of the reasons why a pattern is classified into that category can be found in Appendix 1: Cloud applicability taxonomy of architectural patterns (CATAP).

The discussion that follows shows the different potential ways to explore a pattern's applicability for the cloud environment. The strategies used throughout the research for classifying the architectural patterns are:

- Literature review (illustrated below for the LAYERS and the COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) patterns)
- Review of the available cloud offerings (see LAYERS and COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) sections below)
- Reviewing an architectural pattern with respect to the essential cloud characteristics (see LAYERS, and INDIRECTION LAYER and BROKER sections below)
- Inspecting the relations between architectural patterns (see INDIRECTION LAYER and BROKER sections below)

The usage of these strategies is shown through a discussion of particular architectural patterns. These patterns were chosen so that the strategy could be made clear; also the discussion covers patterns from all three cloud applicability categories containing patterns in them thus covering the different perspectives.

LAYERS

The LAYERS pattern is very simple – decompose the application into logical layers, each containing components on the same abstraction level and allow, in the strict case, only adjacent layers to communicate with each other (Avgeriou & Zdun, 2005). The pattern is easy to grasp and at the same time is very powerful and adaptable. It has been very popular before cloud where the logical separation between presentation, business and data layers was often used, but other possibilities for such "horizontal partitioning" have also been widely used (Buschmann, et al., 2007a, pp. 185-187). Considering cloud, one of its essential characteristics is rapid elasticity – the ability to easily and dynamically adjust based on current needs the resources available to a system running in the cloud; important addition to that is the illusion that this ability is infinite. As Fehling et al. (2014c, p. 6) note there is a shift in the scaling strategy in cloud compared to traditional environments – horizontal scaling (increasing the IT resources) is preferred over vertical scaling (optimizing a system to perform better on the current IT resources). One of the primary reasons is that with vertical scaling the perception of infinite resources at hand is hard to achieve. Having all this in mind, horizontal partitioning (layering) and horizontal scaling are naturally a good combination because independent layers can be scaled independently according to their resource demands – all this increases the importance of the LAYERS pattern; Varia (2011) even discusses architecting for the cloud almost exclusively from a layered perspective. To further confirm this view some cloud PaaS platforms e.g Microsoft Azure (Microsoft Corporation, 2015), Heroku (Heroku, Inc., 2015), AppHarbor (AppHarbor, 2015) even provide the notion of internet-connected and internet-disconnected resources (called web and worker roles, web and worker dynos, web and background workers respectively) – the former are connected to the Internet and serve as a web server processing web requests,

whereas the latter are intended, for example, for intensive, potentially asynchronous business processing and running services in the background like an application server; such a differentiation enforces a layered thinking.

The above discussion of the LAYERS pattern is based primarily on 2 strategies for exploring patterns: literature review and review of the available cloud offerings. Using literature review was already discussed in 3.3.2 Definition of solution objectives. The review of the available offerings, on the other hand, provides an anchor to the current state in cloud computing which often enforces additional constraints to systems. These constraints can be separated generally into 2 subjective categories – cloud infrastructure dependent and cloud paradigm dependent. The constraint falling into the first category are primarily a result of the current state of the technology and may disappear with the evolution of the cloud infrastructure. On the other hand, those constraints falling into the second category are often bound to the essential cloud characteristics and cloud as a conceptual model. Embracing that categorizations of constraints, the above-mentioned cloud platform-specific examples may not stand the test of time, but provide an important perspective – the choice of a cloud provider is important and it has additional implications on the system architecture. After all that said, it is important to note that the discussion of LAYERS above is also bound to and in fact primarily based on the cloud characteristic of rapid elasticity, which makes it vendor-independent.

INDIRECTION LAYER and BROKER

The INDIRECTION LAYER pattern can be thought of as a two- or three-layered system, where one of the layers abstracts the layer beneath it. What this accomplishes is isolation of changes, hiding complexity or heterogeneity of the underlying layer (it may consist even of different systems). In that sense INDIRECTION LAYER can be treated as a variant of the LAYERS pattern which is based on a specific architectural tactic such as using encapsulation and restricting communication paths for achieving modifiability (Bachmann, et al., 2007). Nevertheless, the INDIRECTION LAYER has important implications for cloud and especially for hybrid cloud systems and systems in the process of transfer to cloud. No matter the purpose of an INDIRECTION LAYER it should always hide location, because of the resource pooling cloud characteristic discussed in section 2.1.1 Cloud computing. In that sense the INDIRECTION LAYER may often function as a BROKER considering the Distribution view, and keeping in mind that cloud systems are inherently distributed.

As for the BROKER pattern, the aspect of its solution and applicability has some changes for the cloud. As indicated in the paragraph above, BROKER may be used as an INDIRECTION LAYER and it is very possible this may be the majority of the cases for cloud. The reason for this statement is that the cloud characteristics are often ensured through extensive use of the BROKER pattern in the cloud infrastructure itself and such services are often provided as part of the cloud platform in the form of, for example, elasticity managers and load balancers (Fehling, et al., 2014b, pp. 250-256) enabling the rapid elasticity cloud characteristic. Also in order to provide the resource pooling and broad network

access cloud characteristics, message-oriented communication is extensively used in the form of elastic queues (Fehling, et al., 2014c, pp. 247-259), enterprise service bus and many others – these are realized as cloud platform services and often only configuration is required; Hohpe & Wolf (2003) provide an extensive description of many messaging options. Therefore, from the standpoint of the extensive messaging usage, this brings to front the MESSAGE QUEUEING pattern, which as Avgeriou & Zdun (2005) note often uses BROKER internally, and at the same time potentially reduces the explicit usage of the BROKER pattern. This does not mean that BROKER is not used in a cloud environment but that its usage is rather closely related to the options provided by the chosen cloud platform and hidden inside those. Implementations of the traditional BROKER as described in the pattern are still possible, but the availability of easier and often flexible ready-for-use alternatives and the potential complexity of fully custom solutions based on that pattern reason including the BROKER pattern into the Oblivion category.

The discussion of the INDIRECTION LAYER above is based on comparison to a very similar pattern, LAYERS, and outlining and evaluating their differences. Its evaluation is again based on its compatibility with the essential cloud characteristics. Another aspect of this discussion needs to be underlined – although the pattern itself transfers to the cloud unchanged, there is a benefit in viewing its usage from a cloud perspective as this brings up specific scenarios such as hybrid applications. Documenting this can help practitioners find more easily the potential patterns that may be of use for their case.

Apart from that the analysis of the INDIRECTION LAYER also involves the BROKER pattern and later the BROKER pattern is also discussed with MESSAGE QUEUING in mind. These relationships between the patterns are important. BROKER is potentially used almost exclusively as an INDIRECTION LAYER which means that the 2 views they are part of are linked, thus discussing one pattern may convey much more information than normally. As for the relation between MESSAGE QUEUING and BROKER leads to one pattern taking over another one – the former is extensively used which to great extent limits the explicit usage of the other pattern. Another important point here is the significance of making a difference between cloud infrastructure and a system built on top of that infrastructure. As noted in the section Theoretical Background, such a difference is rarely found in the literature on the subject so far and this example illustrates why it is important to have it.

COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS)

COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) is a new pattern which emerged with cloud. In essence it separates update from read operations thus simplifying the typical CRUD (Create, Read, Update, Delete) data processing model; Figure 8 gives one possible view on a system using CQRS. This opens up many opportunities such as parallelization of read and write operations, increasing decoupling and choosing different scaling strategies for read and write. It is important to evaluate carefully whether or not a system really needs that pattern, because it introduces complexity in terms of resiliency and consistency for

the read operations (Microsoft Corporation, 2014); the latter article as well as (Betts, et al., 2012) and (Fowler, 2011) provide a good overview of the pattern. All the positive characteristics of the pattern fit well into a cloud environment. For example, some cloud providers (Microsoft Azure Storage, Amazon RDS, and Engine Yard, to name a few) offer a service to provide read-only replicas of the data with eventual consistency (more on eventual consistency in (Fehling, et al., 2014c, pp. 126-130). Using these replicas for the Query model part of the pattern offloads the master write database which makes for better user experience and better availability on write operations. On the other hand, if a system expects a large amount of reads but does not require real-time up-to-date data this segregation also improves on responsiveness of the system and allows for optimal scalability.

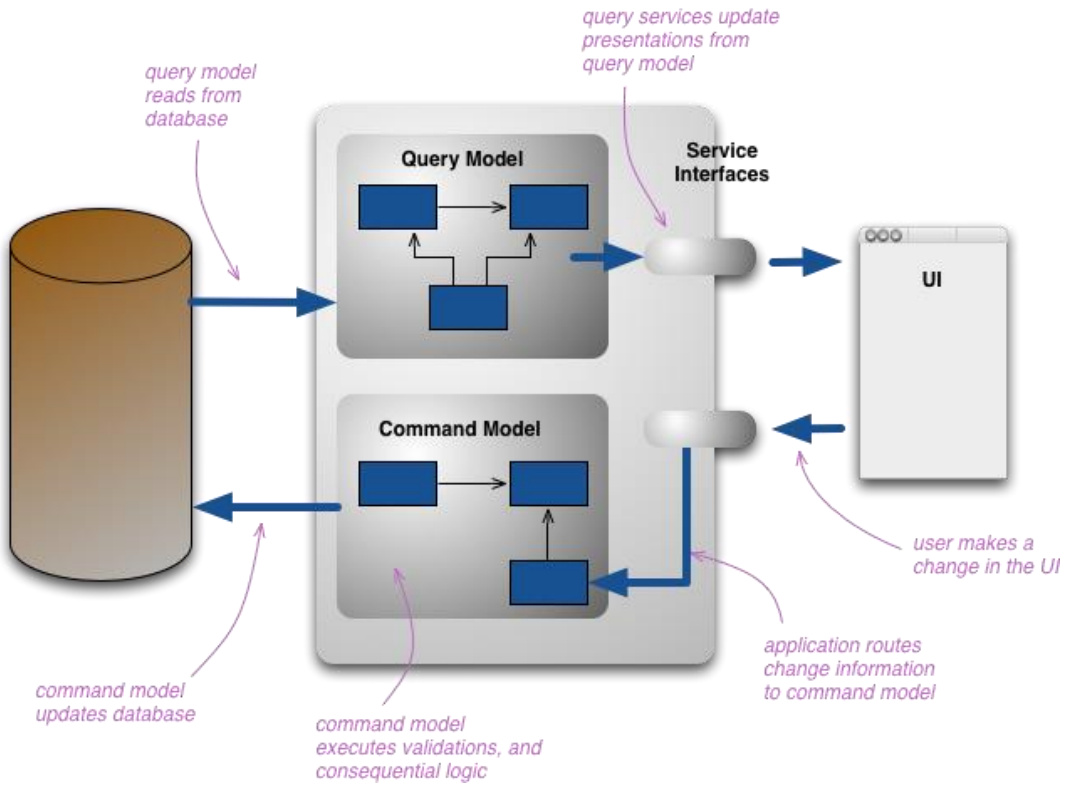

Figure 8 CQRS. Source: (Fowler, 2011).

The details about the rest of the patterns are out of the scope for this report, but the results of their evaluation can be found in the APE Wiki.

4.1.5 Analysis of the CATAP

There are a few more points that need to be addressed in that section. The results can be looked at from two perspectives – first, based on the categories in the CATAP and, second, considering the traditional categorization by views from (Avgeriou & Zdun, 2005).

The CATAP perspective

First of all, there was no pattern identified to belong to the Evolution category. One possible explanation is that the traditional patterns discussed here have been known for a long time, have been documented extensively and have reached a high level of maturity – their definitions are accommodating, flexible and timeless, exactly as Alexander, et al. (1977) defines patterns.

Another observation is the large amount of patterns in the Transfer category. An interpretation of this result can be that it is an indicator of the way cloud has emerged – it is a big step in the evolution of the software development paradigm, but at the same time it originates and builds on the achievements before it e.g. the development level in the fields of distributed computing and service-oriented systems. Therefore, cloud should be perceived as a natural evolution rather than an evolutional jump, a paradigm shift of mature science (Kuhn, 1962) which is still happening; cloud co-exists with the traditional environments thus allowing for reuse of major parts of the knowledge so far. At the same time, all this knowledge needs to be revisited because as the discussions above show there are implications which need to be revealed in order to use the new paradigm to its fullest potential.

Third, despite there is only one new pattern, it is enough of a proof that there is a need to take a closer look at this paradigm shift. Also considering some of the patterns, orthogonal models and tactics provide new perspectives. For example, the patterns SHARED REPOSITORY and ACTIVE REPOSITORY can use Actors concurrency model which recently gained long-awaited popularity (Hewitt, 2015; Corrêa, 2009); one of the examples of a framework built with the ideas of the Actors model and build especially for cloud is presented in (Bykov, et al., 2011) with a case study of cloud system built with it presented in (Bernstein, et al., 2014).

Perspective based on the traditional classification by Avgeriou & Zdun (2005)

Considering the views as in (Avgeriou & Zdun, 2005), there are also some observations which are valuable noting.

First of all, the Adaptation view seems to be not of a too big concern as most of the patterns in it are in the Oblivion category. This is understandable as the cloud environment is homogeneous and predictable.

Going further, the Layered, Data-flow, Data-centred, and User Interaction views all transfer as a whole to the cloud environment – a layered style is considered natural for cloud, Data-flow patterns benefit from the unlimited access to resources and provide good options for scalability, the sole definition of the Data-centred view hints at the applicability and importance of the patterns in it, and most systems have user interface (even more applicable for cloud as access to the machines the software is deployed to is somewhat limited and systems are inherently distributed) which currently tends be a separate loosely-coupled component or even a partially autonomous sub-system with its own architecture. This emphasizes the importance of viewing cloud system architecture from these views' standpoint.

Something more, though, can be observed in the Data-centred view. Although the patterns in it move unchanged they are unsurprisingly dependent on the storage options which are far more than what is typical for a traditional environment (Fehling, et al., 2014c; Wilder, 2012). Different storage options are applicable in different scenarios and sometimes work well with one pattern or another. In addition, the concept of eventual consistency (Fehling, et al., 2014c, pp. 126-130) is rather new and not very typical for a traditional system. Using storage as a service also requires a change in the thinking about system architecture as the communication over network may potentially have much bigger impact on a system.

Contrary to the 3 views outlined above, other views such as the Component Interaction view have patterns in Oblivion and Transfer. Patterns from the same view falling in the same CATAP category often have common characteristics which are the reason for them being categorized in the respective category e.g. the location-agnostic, favouring asynchronicity and low coupling patterns in the Component-Interaction view are transferred to cloud, the rest are deemed not so relevant thus part of the Oblivion category. The next section discusses in more details such relations.

4.2 Patterns-to-characteristics mapping

For this section the perspective shifts from the patterns as a whole to their characteristics and support for particular quality attributes such as scalability, performance, modifiability.

The rest of the section follows the sequence of tasks performed in order to create Artefact 3, the patterns-to-characteristics mapping. First, the process of selecting what characteristics to evaluate the architectural patterns on is presented. After that Artefact 3 is described as a final result; the thinking behind the constructed Artefact 3 is presented with examples for particular architectural patterns in the next subsection. Finally, the section concludes with analysis of the results.

4.2.1 Selection of characteristics

In order to select a set of characteristics that can be used to evaluate the architectural patterns from CATAP on existing credible models can be utilized. Many software quality models defining a set of characteristics with potential measures and relationships have been devised over time. Miguel, et al. (2014) have made an extensive study of a large number software quality models. They reach to the conclusion that many models use ISO/IEC IS 9126 (2001) as starting model and build on it and that "future works will have as main reference this model" referring to ISO 25010 in its 2008 version. In 2011 the ISO/IEC 25010:2011 (2011b) effectively superseded ISO 9126. The new standard defines 2 quality models: Quality in use model and Product quality model. The first model is focused on characteristics oriented towards the human-computer system: effectiveness and efficiency of use, satisfaction, freedom from risk, and context coverage. Some of these have subcharacteristics defined. The same characteristics-

subcharacteristics system applies to the second, Product quality, model as well – there are 8 main characteristics and 31 subcharacteristics distributed across the main characteristics presenting different aspects of them. The Product quality model is applicable to "static properties of software and dynamic properties of the computer system" (ISO/IEC 25010:2011, 2011b). These models are intended to provide a shared vocabulary when talking about quality and one of the activities that can be applied to is "identifying software and system design objectives".

Considering all that said, this paper uses the Product model defined in ISO/IEC 25010:2011 (2011b) as a starting point but the characteristics suggested in the model were carefully analysed for their applicability to the discussion about patterns – Appendix 2: Relevance matrix for the characteristics defined in highlights some of the details in the process including the reasoning behind renaming some of the characteristics in order to reflect the context better. The result is that 15 of the 39 characteristics and subcharacteristics defined in the standard were selected as applicable for this work; 2 of the characteristics including their subcharacteristics were completely excluded (Functional Suitability and Usability) and 2 of the characteristics (Portability and Security) were chosen to represent the applicable part of their subcharacteristics.

Apart from the characteristics defined in the ISO/IEC 25010:2011 (2011b) Product model, a few other characteristics stood out during the research and were mentioned in the paper so far. Some of them were already used as contextualizing replacements for some of the characteristics and subcharacteristics in the model defined in the standard – Scalability instead of Capacity in the context of Performance, Multi-tenancy instead of Co-existence in the context of Compatibility. As a side note, Scalability is the term used in the tables, although the right term for cloud should be Elasticity – dynamic scaling. The reason for using the former terms is because Elasticity is not typical for traditional systems and is much harder to achieve thus discussing it for patterns used in a traditional environment is not proper.

Location transparency is one of the characteristics which deserves some more attention, although it is slightly inconsistent with the rest of the characteristics as it is related to a quality aspect of the architecture rather than a product quality characteristic. Location transparency is mentioned in (Avgeriou & Zdun, 2005) in the context of the Distribution view, but is also in the focus for the Component Interaction view as the communication is in some patterns dependent on a location identifier; Buschmann, et al. (2007a) also pay significant attention to it especially in the chapter on interface partitioning (2007a, pp. 271-303).

Cost is another aspect which is important in the case of cloud architecting. As the cost is recurring and because it is based on resource usage software architecture plays a very important role for the cost optimization. Pramod, et al. (2013) point out that cost analysis for cloud is more complicated than analysis for a traditional environment. It is important to take in mind data transfer, processing power, and vendor and cloud service model specific details. They further highlight that developers and architects need to carefully consider the trade-offs of the different costing models in the context of the requirements and this inevitably affects the

architectural decisions. Although all this shows that cost is an important factor to be considered by architects, it is not possible to include it in the discussion of this research as vendors have many and different cost models and cost apart from influenced by architectural decisions it also depends significantly on implementation details and particular technology choices.

4.2.2 The patterns-to-characteristics mapping

Before going into details about Artefact 3, an important limitation needs to be outlined. The patterns in the User Interaction view of (Avgeriou & Zdun, 2005) are often very independent of the cloud environment – the user interaction is to various extent distributed across cloud and non-cloud (e.g. mobile devices, browsers) components. Important decisions need to be taken with respect to these pattern which influences the functionality of the cloud-hosted part of the system, but at the same time these patterns are flexible and are only indirectly related to what the cloud has to offer. This is the reason they will not be further discussed in this paper and are not included in Artefact 3.

Pattern	LAYERS	INDIRECTION LAYER	MESSAGE QUEUING	CQRS	BROKER
Cloud category	*Transfer*	*Transfer*	*Transfer*	*Innovation*	*Oblivion*
Characteristic					
Performance	–	–	–	–	–
Elasticity (Scalability)	▲	–	▲	–	–
Multi-tenancy	–	–	–	–	–
Interoperability	–	–	–	–	–
Availability	▲	–	▲	–	–
Fault-tolerance	–	–	▲	–	–
Recoverability	–	–	▲	–	–
Modularity	–	–	–	–	–
Reusability	–	–	–	–	–
Analysability	–	–	–	–	–
Modifiability	–	–	–	–	–
Testability	–	–	–	–	–
Portability	–	–	▼	–	–
Security	–	–	–	–	–
Location transparency	–	–	–	–	–

LEGEND			
Traditional environment		Cloud Environment	
	support	▲	enhance
	neutral	–	no change
	hinders	▼	diminish

Table 2 Patterns-to-characteristics mapping (excerpt)

Appendix 3: Patterns-to-characteristics mapping contains the full matrix of patterns observed in the context of their characteristics during the research presented here and in fact represents the aimed Artefact 3. Due to its size, a summarized excerpt of it containing the patterns mentioned so far in the report is presented in Table 2.

Both tables use a 2-scale system to represent the information – background colour and icon convey different streams of information.

The colour of the cells represents how the patterns relate to the respective characteristic in a traditional environment. The results presented are primarily based on literature - (Buschmann, et al., 1996), (Buschmann, et al., 2007a), (Buschmann, et al., 2007b), (Clements, et al., 2010), (Rozanski & Woods, 2011) to name a few. The red colour marks that the pattern under consideration affects negatively the given characteristic; the yellow colour shows neutrality towards the characteristic or a balanced mix of benefits and liabilities; the green colour means the pattern helps achieve the specified characteristic in a system. This three-valued scale is similar to the one used in (Buschmann, et al., 2007b), but takes on patterns from another perspective. A five-valued scale with values "hinders", "may hinder", "neutral", "can enhance", "enhances" could arguably provide some more information to the reader, but makes the table less readable – therefore the compromised 3-value scale was chosen. Apart from that, the table itself should not be considered a definitive guide, but just a starting point and the details about patterns need to be read – leaving the reader with a less-detailed information in that case can even be taken as beneficial for the purpose.

The changes with respect to cloud are represented with a set of well-established icons depicting a change in a negative direction, no change or a change in a positive direction. The change in a negative direction means that, for example, if the pattern supported strongly a characteristic in a traditional environment, for cloud this support is not that strong because of particular traits of the cloud context; the same type of logic applies for the rest of the icons. It is important to note that having up or down arrow does not mean changing the level of support completely e.g. from supporting to neutral or even hindering – the changes identified have been rather minor.

4.2.3 Implications for a sample of architectural patterns

The sample presented in Table 2 serves as a foundation for the discussion in the following paragraphs illustrating the research process during this stage of the study.

LAYERS

Most often the LAYERS pattern is discussed in terms of enhancing scalability, modularity, reusability, portability, and analysability and modifiability in terms of localizing potential changes (Buschmann, et al., 1996; Clements, et al., 2010, p. 89). The usual liabilities of the pattern are related to performance and the risk of cascading changes which negatively impacts modifiability and analysability (Buschmann, et al., 1996; Clements, et al., 2010, p. 89); analysability is further

negatively impacted by the potential complexity of the communication. The dual nature of modifiability and analysability – both as potential benefits and risks – reasons marking them with neutral i.e. the pattern has a contradictory behaviour with respect to them. Looking at the pattern in a cloud environment, the modularity of the pattern and the well-defined communication paths fit well in the rapid elasticity cloud characteristic. The flexibility of the pattern allows systems utilizing it to scale both horizontally and vertically thus the LAYERS pattern becomes even slightly more favouring the scalability (elasticity) characteristic; the trend of building stateless components for cloud (Fehling, et al., 2014a, pp. 171-175) allows the benefits to be even clearer. Essentially all this also influences positively availability, because if a part of dynamically scalable layer is not available at a given point in time, using dynamic elasticity a substitute of that part should be available and eliminate any downtime.

INDIRECTION LAYER

As for the INDIRECTION LAYER, the differences with the LAYERS pattern are clearly outlined in Table 2. Because of the function of the INDIRECTION LAYER to primarily provide interoperability and abstraction, the characteristics of interoperability and location transparency are better supported even in a traditional environment, whereas analysability may become a bit trickier. Generally speaking, as the INDIRECTION LAYER can be thought of as a more specific and prescribing variant of the LAYERS pattern, these observations are expected and natural.

MESSAGE QUEUING and BROKER

The MESSAGE QUEUING pattern was so far mentioned in comparison with BROKER and based on the reviewed sources MESSAGE QUEUING is favoured in a cloud environment. Comparing both patterns in terms of their characteristics in Table 2 sheds some more light on why this is the case. Essentially, MESSAGE QUEUING provides better support for fault-tolerance, recoverability and trails on analysability and security. Although MESSAGE QUEUING does not support the former 2 characteristics, the technical solutions used in implementing it usually support some kind of security, whereas analysability can be achieved with relatively little effort. Furthermore, as it will be discussed shortly, these 2 characteristics seem to be of less significance for cloud adaptation.

CQRS

Going to CQRS as a representative of the Innovation category, having a cloud and traditional comparison is of no use, because the pattern has been specifically designed for cloud and using it in a traditional environment in most cases should not be advisable. For consistency, the pattern's relation to characteristics is described using the same representation mechanism. Because the pattern has a relatively high number of liabilities such as decreased fault-tolerance, recoverability and reusability, it needs to be carefully considered whether the scenario will benefit from it (Fowler, 2011). On the bright side, recoverability can be achieved on the implementation level by using the Event sourcing pattern, which is covered

in (Fowler, 2011), and is especially relevant when building stateless components. Apart from that, CQRS is one of the few patterns for cloud which affect performance positively and in addition it also has very good capabilities for availability.

4.2.4 Analysis

Looking at the overall results of this part of the study (see Appendix 3: Patterns-to-characteristics mapping), almost all of the patterns in the Transfer category support better at least one of the investigated characteristics in a cloud environment and none of the patterns in the Oblivion category has been found to provide better support in a cloud environment for any of the characteristics.

Further, 5 characteristics receive an increased support in cloud by patterns: scalability, location transparency, availability, recoverability, and fault-tolerance; for brevity they will be referred to as SLARF for the rest of this section. This set of characteristics is unsurprisingly well-suited for the cloud environment. Availability, recoverability, and fault-tolerance are all related to the fact that cloud computing environments are run by a third party and are in general less reachable by support and administration staff of the cloud consumer thus the need for more "intelligent" and self-sustainable systems which can cope with exceptional conditions. Scalability and location-transparency were discussed several times in this paper and the findings re-confirm their importance.

Figure 9 presents in relative measures (without a scale) the trends for the support of the characteristics in the Transfer and Oblivion pattern categories. The Innovation category was not included as currently it contains only one pattern which does not provide good grounds for further analysis.

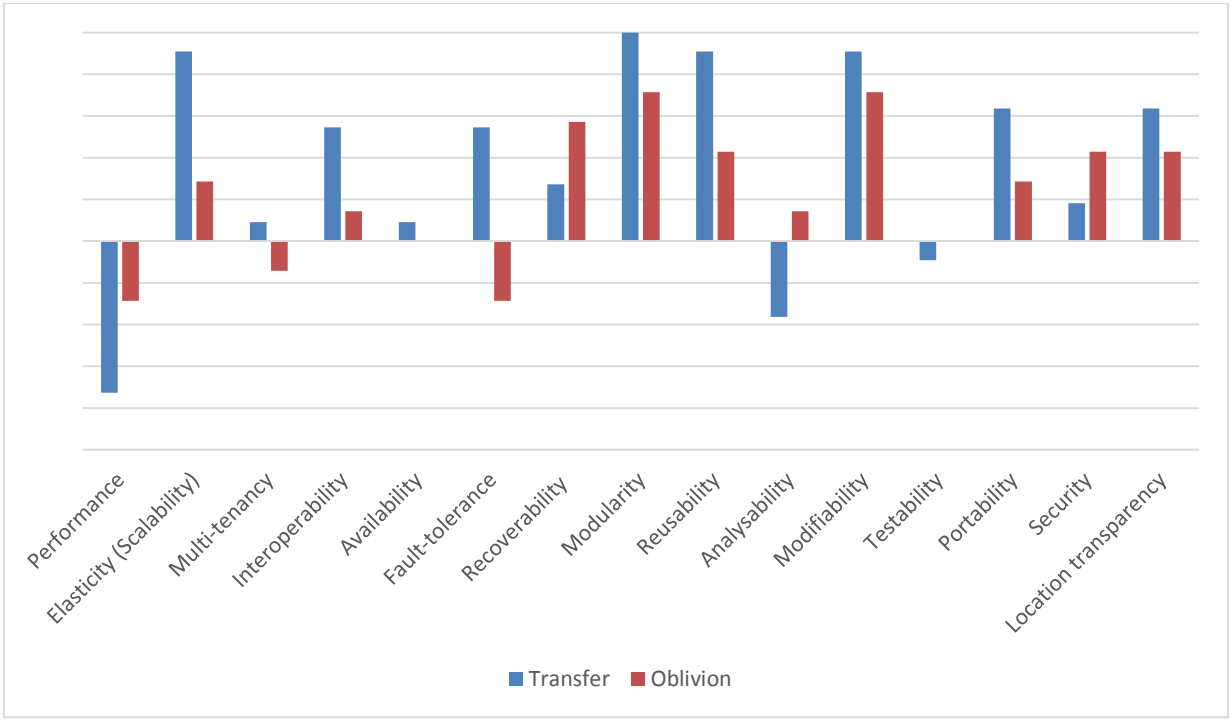

Figure 9 Pattern support for characteristics by cloud applicability categories.

First of all, it seems that very few patterns overall of those which are part of the study favour performance; most of them fall into the Transfer category. This does not mean that performance is not a desirable attribute, even the opposite – a requirement for high performance is one of the most common requirements for every system. Considering ISO/IEC 25010:2011 (2011b), the performance characteristic there encompasses both the performance and elasticity characteristics as used in this study. So the lowered expectations towards time behaviour and resource utilization are compensated with higher expectations towards scalability i.e. achieving performance benefits through horizontal scaling rather than optimizations.

Although multi-tenancy was not one of SLARF, patterns in the Transfer category seem to slightly favour it. The initial expectation in the study was that this difference would be much more significant, because as the cloud platform itself, the tendency in cloud applications seems to be clearly towards multi-tenancy. Therefore, ensuring multi-tenancy is to a bigger extent a concern for the actual implementation than it is in the pattern choice.

Availability, fault tolerance, location transparency, interoperability, modularity, reusability, modifiability, portability and are all well represented in the patterns in the Transfer category with good margin compared to their presence on average in patterns in the Oblivion category. Availability, fault tolerance and location transparency were expected as they are part of SLARF. Cloud is very much about integrating modular subsystems and third party systems and components so interoperability is also not surprising; this is very similar to service-oriented architecture which authors often compare it to (Erl, 2005; Erl, et al., 2014). Modularity follows logically with even greater importance as achieving scalability is to a large extent dependent on having a well-defined and discrete components which can individually be horizontally scaled; modifiability is to a significant extent influenced by this as well. Portability comes as a small surprise, though. Portability in the sense of cloud platform switching is rather hard to achieve currently with the tools at hand because of the lack of standardization of cloud vendors potentially leading to vendor lock-in – this is a valid concern expressed by practitioners but it is still to be acted on (McKendrick, 2011; Pramod, et al., 2013). On the other hand, portability in the sense of deployability is very desirable as the scale at which cloud systems can potentially operate is rather big – in that case the versatility coming from patterns should be of good use.

Another surprise, recoverability although part of SLARF is better supported by patterns in the Oblivion category compared to those in the Transfer category. Possibly this is due to trade-offs with other characteristics, but at the same time it is a characteristic that is supported by the cloud environment. Apart from that, this also opens a niche for improvement. Potentially some of the patterns in the Oblivion category, some of which were even on the verge of Transfer can be considered in order to enhance recoverability, or new patterns may emerge.

Finally, analysability, testability, and security are also all better represented in the Oblivion category. The same considerations as expressed for recoverability can be applied here. Very few patterns of those examined actually support analysability

and they are all in Oblivion. This is understandable as the patterns which provide pre-defined and easier to follow ways of communication were filed as part of the Oblivion category – a consequence of putting emphasis on location transparency, modularity and elasticity. This means that special attention and extra effort during the implementation needs to be put into system aspects such as telemetry, logging and traceability. The case for testability and security is very similar but with a smaller margin between Transfer and Oblivion is not that big. As a side-note for security, it has been often cited as one of the major stoppers for adopting cloud in enterprises. Because of that, significant effort has been spent by cloud vendors to provide stable secure infrastructure which should help in achieving better architectural security, but it should still be an important concern which needs special attention and rigorous engineering processes.

All these observations are a consequence of analysing the CATAP categories of patterns as a whole. Therefore the results presented can provide food for thought, but considering the patterns one by one in concrete is the way to go when designing a system.

Architectural patterns' applicability for cloud and their characteristics were discussed so far. The next sections present the next step in the research: documenting the findings.

4.3 Pattern form

The pattern form provides a template for describing a pattern. The results found so far need to be externalised in a structured and standard manner, which motivates the usage of a pattern form. Due to the focus on the effects on patterns coming from the cloud environment, the existing pattern forms need to be evaluated for their suitability and potentially extended to suit the goals of this research. This section first discusses the requirements that drove the design process at this stage, then an overview of the available options is presented and finally, two pattern forms which result from the work during this stage of the research are described in details.

4.3.1 Requirements

For the purpose of the research a few requirements were defined to drive pattern form construction. These requirements are based on the research objectives and the overall aim of the thesis work. The requirements are:

PF_REQ1. The relations between the architectural patterns need to be explicit and clear.

PF_REQ2. The characteristics affected by the pattern and the particular effects need to be clearly documented as part of the pattern form.

PF_REQ3. The pattern form needs to be kept minimal, but the relation to the architectural pattern in general has to be clear.

Considering PF_REQ1, the relations between the patterns need to be made as clear as possible. The context of the relationship whether it is cloud-specific needs to be obvious. Therefore and considering the focus of the report is on the cloud environment, a differential approach will be taken – only differences compared to the usage of the pattern in traditional environments will be externalized. Of course, it makes sense to have the knowledge gathered about patterns looked from different perspectives along the years organized into an integral body of knowledge, but such an enterprise is out of the scope for this work.

As for PF_REQ2, it is important to have the characteristics influenced by patterns well highlighted in the pattern form. The same explicit documentation of whether a characteristic is specifically influenced in a cloud environment or it was inherited from the generic pattern description needs to be present as well. Outlining these differences highlights what may need to be changed in a practitioner's mind in order to apply the pattern successfully.

Finally, PF_REQ3 presents the need to highlight the cloud-related pattern properties by keeping the resulting pattern form minimal but at the same time without omitting essential information that may hinder getting the whole picture. Despite the previously stated requirement to outline the cloud perspective, a practitioner looking at the pattern documented using the proposed form should still receive enough information about the pattern. This may be done through referencing and redirecting to other sources and summarizing important pieces of knowledge about the pattern.

4.3.2 Discussion of options

Considering the requirements stated above, a natural choice is to use the common sections identified by Rozanski & Woods (2011, p. 162): name, context, problem, solution and consequences.

The name will be the name used in (Avgeriou & Zdun, 2005) but as they note some sources use different names, so such information, if applicable, will also be included.

The pattern context in general should shortly be presented. Another paragraph should outline the cloud-specific considerations.

The problem should follow the same principle as the context.

The solution should be presented in general, but it should also include hints at potential cloud offerings which can be beneficial for implementing a concrete system. Although this second part will not be "timeless" (Alexander, et al., 1977), it may serve as an important guideline for architects.

The consequences section should have up to 3 sub-sections. The first section should discuss positive influences in terms of characteristics as shown earlier in the section Patterns-to-characteristics mapping. The second section should address the negative influences following the same principle. The last section should provide a narrative of other aspects of the results from applying the pattern.

In addition to these sections based on (Rozanski & Woods, 2011, p. 162), one last section will be added which addresses the requirement for making the relations between patterns explicit. The inclusion of such a section resembles slightly the Next section of the form used by Hohpe & Woolf (2003, pp. xli-xliv) and is also inspired by it. The narrative in this section should be similar to the discussions presented earlier in the section Cloud applicability taxonomy of architectural patterns.

4.3.3 A full pattern form (FPF)

Based on the 3 requirements towards the pattern form and the options discussed above, a full pattern form (FPF) was first designed. The FPF captures all the information about the architectural pattern, primarily only extending an almost standard pattern form with well-defined cloud sections. The fact that it combines a lot of information – both environment-specific and patter information in general – is the reason for calling it a "full pattern form".

The FPF is captured as a template on Figure 10. The pattern form sections in the figure are in **bold**. Subsections are increasingly indented to show visually which the parent section they belong to is. An explanatory text is given in *italic*.

The FPF is definitely not minimal which contradicts PF_REQ3 – the pattern form needs to be kept minimal. Having that separation, though, allows for extracting a shortened pattern form which can be used to depict only cloud considerations and differences appearing in cloud – this abbreviated pattern form is in fact Artefact 3 for this research. This abbreviated form is presented in the next section.

Name: <*Primary name. In this thesis, this is the name used in (Avgeriou & Zdun, 2005). For architectural patterns documented out of this study, a primary name should be agreed on*>

Also known as: <*alternative names currently used in literature. The primary name should be used in general after being agreed on*>

Context <"*sets the stage for the pattern, explains its motivation and rationale, and describes the situations in which the pattern may apply" (Rozanski & Woods, 2011, p. 162)*>

General

<*Place here the context of the architectural pattern in general; usage of existing literature is encouraged*>

Cloud considerations

<*Place here any cloud-specific considerations and discussions*>

Problem

Overview

<*Describe the problem here; usage of existing literature is encouraged*>

Standard requirements (forces)

<*Describe here typical conditions that need to be met in order to consider applying the described pattern; usage of existing literature is encouraged*>

Cloud considerations

<*Place here any cloud-specific considerations and discussions*>

Solution

Overview

<*Describe the solution in general here – usually this is a model of the elements of the pattern and their interrelations; usage of existing literature is encouraged*>

Cloud implementation suggestions

<*Briefly present here potential cloud offerings which can be beneficial for implementing a concrete system using this pattern*>

Consequences <*Show what the results from applying the pattern are*>

Positives

<*Include here the positive consequences in the general case in terms of characteristics*>

Cloud

<*Place here any additional positive consequences, if any, appearing because of the cloud environment*>

Negatives

<*Include here the negative consequences in the general case in terms of characteristics*>

Cloud

<*Place here any additional negative consequences, if any, appearing because of the cloud environment* >

Other considerations

<*Place here any other considerations about consequences. This may include neutral characteristics or characteristics with a mixed impact*>

Cloud

<*Place here any other consequences, if any, appearing because of the cloud environment*>

Relations to other patterns

<*Place here a short discussion of architectural patterns which are, for example, complementary to this one or are not suitable for usage with this one*>

Figure 10 A full pattern form (FPF)

4.3.4 A cloud abbreviated pattern form (CAPF)

The third artefact in this research is called a cloud abbreviated pattern form (CAPF); "abbreviated" because it is based on the FPF and "cloud" because it is focused only on cloud-specific considerations and almost nothing more.

The CAPF is depicted on Figure 11. The sections in the CAPF are mapped to sections in the FPF; the mapping uses the section headings as path navigation separated by the # (hash) symbol.

Name: *<Mapped from FPF#Name. It should be clear that this is a not a full pattern description – "(cloud)" could be appended to the name and a reference to the full description of the pattern should be made.>*

Also known as: *<Mapped from FPF#Name#Also known as>*

Context

<Mapped from FPF#Context#Cloud considerations>

Problem

<Mapped from FPF#Problem#Cloud considerations>

Solution

<The generic solution or just parts of it could be included here >

Implementation suggestions

<Mapped from FPF#Solution#Cloud implementation suggestions>

Consequences

Positives

<Mapped from FPF#Consequences#Positives#Cloud>

Negatives

<Mapped from FPF#Consequences#Negatives#Cloud>

Other considerations

<Mapped from FPF#Consequences#Other considerations#Cloud >

Relations to other patterns

<Mapped from FPF#Relations to other patterns>

Figure 11 Artefact 3, a cloud abbreviated pattern form (CAPF)

To illustrate the proposed pattern form the LAYERS pattern is documented in Appendix 4: LAYERS pattern documented using Artefact 3, a pattern form.

The CAPF could exist on its own, but the relation to FPF is important as it shows the knowledge transfer – from the more general pattern description, to the more concrete pattern description taking into account the specifics of the cloud environment.

4.5 APE Wiki

Wiki pages as a tool for knowledge sharing were invented by the pattern community to facilitate the discussion of patterns (Buschmann, et al., 2007b). In that spirit a wiki, the APE Wiki is the last artefact constructed as part of the Design and Development stage of the research.

The following sections discuss consecutively the requirements imposed on the wiki engine to be used, the choice process of a wiki engine and, finally, some details of the concrete implementation, primarily focusing on information architecture.

4.5.1 Requirements

The choice of a wiki engine was driven by 4 requirements, all of them elicited on the base of the domain formed during the research for the previous three artefacts. The requirements are:

APE_REQ1 Support for tables – the option to create a table in the wiki is required as tables, as demonstrated in this paper, allow for providing a well-structured, visual and to some extent detailed overview of a set of objects of interest e.g. patterns and characteristics.

APE_REQ2 Support for categories – the categories in a wiki are a way to connect different related articles; it also provides generic navigation options for articles within a category – (Wikipedia, 2014a) is an example of such a page.

APE_REQ3 Referencing support – as the APE Wiki is an artefact part of a scientific research, it must use rigorous referencing. Apart from that, the presence of reference support provides links to both scientific and non-scientific valuable articles so that the results from the study are connected to the existing knowledge base.

APE_REQ4 Support for tables of contents – patterns are presented in a structured way, so a table of contents for providing navigation capabilities is a "should have" requirement.

APE_REQ5 Ease of deployment and administration - as the aim of the artefact is to effectively present the results, the deployment and administration of the wiki engine should not be a hassle; therefore the wiki engine should provide the above requirements introducing minimal administrative overhead.

4.5.2 Discussion of options

The pattern community maintains an extensive list of existing wiki engines (2014). Based on the requirements for the APE Wiki stated above and with help from the pattern community wiki choice tree (2010) some of the more prominent wiki engines were evaluated. Table 3 presents a high-level overview of well the 4 wiki engines considered in more details match the requirements.

	TiddlyWiki	**Gollum**	**TikiWiki**	**MediaWiki**
APE_REQ1	Yes	Yes	Yes	Yes
APE_REQ2	Partial (tags)	Yes	Yes	Yes
APE_REQ3	Yes, 2 plugins	Yes	Yes	Yes
APE_REQ4	Plugin	Yes	Yes	Yes
APE_REQ5	Hard, custom development	Medium	Medium	Easy

Table 3 Evaluation matrix for wiki engines.

Based on Table 3, MediaWiki is the obvious choice. In fact, MediaWiki and TikiWiki were much closer considering them in details. For example, TikiWiki has a very agile referencing system, whereas MediaWiki is much easier to administer primarily because TikiWiki has also a very agile, but requiring permission-based model of access. Also, MediaWiki is the most popular wiki engine which powers Wikipedia, for example, and this makes it more familiar to the end user. Because of these considerations, MediaWiki is the wiki engine of choice for constructing the APE Wiki.

4.5.3 Implementation

The latest stable version of MediaWiki, 1.24.2 at the time of writing, was deployed to a hosting provider. The APE Wiki can be found on http://patterns.freehosting.bg/mw.

The APE Wiki is organized using a left hand side menu and using categories. The left hand side menu provides a quick way to go to the part of the APE Wiki that is needed. Some of the options there are links to category pages which act as a catalogue for the category e.g. every cloud applicability category is represented in the menu and the link points to the category page for the respective cloud applicability category. Other pages present through text the information; for example, the page on the CATAP uses the table used in this report as a starting point and gives some background on what the CATAP is.

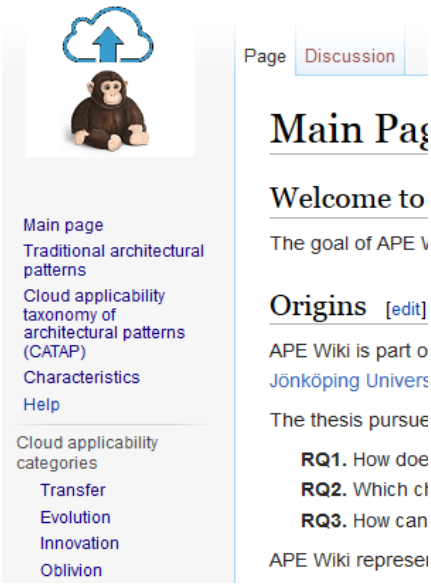

Figure 12 APE Wiki: left hand side navigation

MediaWiki templates help ensure consistency and compliance to the pattern form. The pattern as a whole, using the full pattern form (FPF) shown on Figure 10The FPF is definitely not minimal which contradicts PF_REQ3 – the pattern form needs to be kept minimal. Having that separation, though, allows for extracting a shortened pattern form which can be used to depict only cloud considerations and differences appearing in cloud – this abbreviated pattern form is in fact Artefact 3 for this research. This abbreviated form is presented in the next section.

, is presented on the page http://patterns.freehosting.bg/mw/index.php/Template:Pattern. Then, the relevant information on that page is used to create 2 pages presenting a subset of the form – one page is concentrated on the architectural pattern from a traditional perspective and another page focuses on the cloud perspective; the latter uses the form defined in Artefact 3 defined in the template that can be found on http://patterns.freehosting.bg/mw/index.php/Template:Pattern_cloud.

To ensure consistent referencing, another template has also been defined – http://patterns.freehosting.bg/mw/index.php/Template:RefDef.

Finally, all characteristics discussed so far are also described in separate pages as considering their important role in describing the architectural patterns their meaning must be as clear as possible.

Due to hosting restrictions some of the ideas regarding the APE Wiki could not be implemented e.g. Infobox (Wikipedia, 2014b) and advanced citation mechanisms.

Another important note is that at the time of writing of this report, APE Wiki did not contain all the information gathered during the research. This is due to the time limitation for performing thesis defense.

5 Demonstration

The Demonstration activity is centred on using the artefacts produced in the Design and Development stage. The aim is to provide a natural step for evaluation and iteration of the design science process.

The following subsections discuss, first, how demonstration was carried out in this research and, second, potential strategies for extension and improvement of the process.

5.1 Demonstration in this research

Essentially 2 ways of demonstration are used in the research – demonstration through creation, and the APE Wiki.

Demonstration through creation can be thought of as a special kind of demonstration, because the artefacts' information is to a significant extent based on actual usage. In particular, case studies – descriptions of real-life systems' architecture in online resources such as technical blogs (Netflix, 2015) and also scientific resources (Hu, et al., 2013) – largely contributed to the creation of the main and supplementary research artefacts. In that sense, the link between the demonstration through usage and the artefacts created is reverse, but it builds credibility and is motivated by the nature of the study.

The design science process emphasizes the importance of the Demonstration stage as it is the bridge to iterative improvement of the created artefacts. The APE wiki apart from a central artefact is in fact also the primary Demonstration activity – basically, the APE Wiki is the media showcasing all created artefacts and its aim is to reach to the practitioners – the biggest group of evaluators for the utility of the artefacts. The APE Wiki is an effective demonstration strategy as it is an available to the public online artefact. The implementation of that strategy, though, was limited due to insufficient time – the biggest limitation is in the area of gaining community engagement.

5.2 Options for extension

In the scope of this thesis work the demonstration is somewhat limited by a number of factors ruling out several potential demonstration strategies or implementation details. The next paragraphs discuss the most prominent of these strategies in order to outline potential pathways for improving the research.

The most natural option for extension is actively engaging the pattern community. An example, which could directly use the results of this paper, is to publicize APE Wiki which fell out of the scope of this thesis work due to time limitations. The opportunity to actively engage the community through the APE Wiki is extremely valuable, considering that constructive communication through wiki pages is natural for the pattern community. For the pattern community to actively improve the artefacts and to share thoughts on different aspects of them through APE Wiki facilitated discussions are just examples of the possibilities APE Wiki

provides. All in all, actively engaging the community ties in the Demonstration activity with the Evaluation activity and provides a direct path to design science process iterations through feedback. Even neglecting the initial effort of attracting the community's attention, such an enterprise would require a dedicated and active way of communication with the community which, unfortunately, was not feasible in the scope of the thesis work. Nevertheless, this is undoubtedly the best option for demonstration, validation, and evaluation of the results and a natural bridge to making the research process implementation stronger. First of all, though, APE Wiki at the time of the writing of this report, APE Wiki does not contain all information gathered during this research and this needs to be addressed first.

Another candidate for a demonstration strategy is to use a field study. An example would be to take a number of professionals familiar with the traditional architectural patterns, assign to them a real-world problem that they need to solve through architecting a cloud system by learning about the cloud way of architecting through the artefacts of this thesis. The problem with this approach is that it requires a lot of resources – the professionals would spend time doing that, the resulting system needs to be evaluated for its qualities and how the usage of the artefacts from this thesis affected the resulting system and the knowledge of the professionals creating it.

6 Evaluation

The Evaluation activity deals with "how well the artifact supports a solution to the problem" (Peffers, et al., 2007-8). The Evaluation also ensures the iterative nature of the research process.

The following subsections discuss, first, how evaluation was carried out in this thesis work and, second, discussion of ways the Evaluation stage can be extended and improved.

6.1 Evaluation in this research

Evaluation in this research was carried out along the following 3 axes:

1. Evaluation based on the research objectives;
2. Evaluation of reliability of results;
3. Extensibility and replication of the research process.

These three evaluation axes are discussed in more details in the rest of this subsection.

6.1.1 Evaluation based on the research objectives

First of all, Peffers, et al. (2007-8) suggest that the Evaluation stage, depending on the problem could potentially include comparison of the artefact's functionality with the objectives defined during the second stage of a design science research. Such a comparison is deemed relevant for this research as the research process is intended to provide utility with respect to the objectives defined. Further, considering that 4 inter-connected artefacts were designed during the Design and development stage, an evaluation of these artefacts with respect to the research objectives is important to provide reality check and to ensure the direction of the research is as intended.

Stage 2 in the research process, Definition of solution objectives, identified the following 4 objectives:

RObj1. to pay attention to the reasons why a pattern has transferred, transformed, emerged or became less relevant with respect to the characteristics of the patterns and their suitability for the cloud due to these characteristics;

RObj2. to distinctively present and outline these changes, so that it is clear what the differences are;

RObj3. to allow preservation and inclusion in the pattern description of the information about patterns that has been developed throughout the years and is still relevant;

RObj4. to keep the solution close to practice.

RObj1 is directly related to the construction of the first 2 artefacts, the CATAP and the patterns-to-characteristics mapping. Appendix 1: Cloud applicability taxonomy of architectural patterns (CATAP) provides details about why a pattern

has been classified in a particular cloud applicability category; the discussion in section 4.2 in this report dedicated to constructing the patterns-to-characteristics mapping also provides analysis from the perspective of characteristics; the APE Wiki contains details of the findings of the research.

As for RObj2 and RObj3, using the learnings from already existing pattern forms and extending one such form helps in achieving these 2 objectives the resulting CAPF is an extension to an existing pattern form. Apart from that, the APE Wiki further makes this information publicly available and enables community feedback thus iterations over the research results.

Finally, RObj4 is focused on the utility of the results. Presenting the results in a wiki is arguably one of the most efficient methods to achieve this – the wikis are close to the pattern community (they were invented by the pattern community (Buschmann, et al., 2007b) and a wiki is publicly available and open for collaboration by the community. Another way to ensure RObj4 is the form of presentation. The results are often generalized and condensed in a table or a figure, e.g. Appendix 1: Cloud applicability taxonomy of architectural patterns (CATAP) and Appendix 3: Patterns-to-characteristics mapping – this presentation is suitable for reference purposes for practitioners as a "cheat sheet".

Considering these points, it can be concluded that overall, the constructed artefacts and the research process follow the directions set in the research objectives.

6.1.2 Evaluation of reliability of the results

In order to evaluate the reliability of the results, this research follows 2 principles: ensure rigour in the research and iteratively re-evaluate the artefacts constructed.

Hevner & Chatterjee (2010a, p. 18) highlight the importance of relying on past knowledge in order to ensure innovation of the results and contribution to the knowledge base, instead of simply performing "routine design". Considering this research, it recognizes this trait of design science research. An extensive amount of literature was reviewed and was included in the discussion throughout this report. Furthermore, the sources used were of different origins – scientific and non-scientific sources including books, conference reports, technical and personal blogs, technology manuals, wikis and other community-based efforts. This means that different perspectives were taken into account.

Apart from the significance of the sources, evaluation was part of the process during the design of the artefacts as shown in the section Analysis of the CATAP, for example. On the other hand, subsequently designed artefacts also assisted in evaluation of the artefacts they are based on. For example, while implementing the APE Wiki if an artefact was hard to present in the wiki format or position in the information architecture, then this artefact or the information architecture were re-considered as these are issues hindering achieving the research objectives. The results of such evaluation are either intrinsically captured in the final results or explicitly documented in this report thus manifesting the descriptive evaluation method used in the research. This example also illustrates the internal re-iterations performed during artefact construction.

Finally, following the design science research process also helps in achieving rigour as the stages and guidelines highlighted by Peffers, et al. (2007-8) and Hevner & Chatterjee (2010a) promote the scientific research perspective and critical thinking at all times.

6.1.3 Extensibility and replication of the research process

This paper attempts to document in a rather extensive manner the steps and the guiding principles which would be needed in order to reproduce the research. Ensuring the research process is well documented is not an evaluation by itself, but is a fundament to replication which would in turn provide evaluation – this is the reason why extensibility and replication of the research process are listed here.

Apart from that, documenting the process proved to be an important tool for evaluation of the results through the process perspective as externalizing the process poses a number of questions uncovering potential weaknesses. A consequence of this practice is, for example, the following subsection discussing potential process improvements and extensions.

6.2 Options for extension

Similar to the Demonstration stage, a number of factors limiting the scope of this research affected the evaluation performed. The following paragraphs try to outline some of the options available for extending the Evaluation stage – these options were considered during the research process, but could not be included in the research itself. These options are concentrated on the external validity of the results as a gap in this research which needs to be addressed.

As Hevner & Chatterjee (2010a, p. 19) note, "artifacts must be rigorously and thoroughly tested in laboratory and experimental situations before releasing them into field testing along the relevance cycle". Considering this view, the option of performing a field study, as already discussed in Demonstration, could very well fit into the Evaluation stage.

The other option was also already discussed in the Demonstration section – involving the community to evolve the APE Wiki and therefore the results of the research. This option would allow for a much better implementation of the second principle of evaluation of reliability of the results – iterative re-evaluation.

Both these options are in fact observational evaluation methods. Strengthening these methods and considering the descriptive methods (discussion of scenarios, informed arguments) are already extensively used, this triangulation would overall improve the Evaluation stage of the research.

Finally, how each pattern has been evaluated against the characteristics in Artefact 2 can be elaborated and made explicit, so that to strengthen the replicability of the process. This means providing a set of criteria or questions that should be asked for each of the characteristics and a scale for evaluating the effect of the pattern under consideration.

7 Discussion and Conclusions

This paper looked into how the knowledge about software architectural patterns reacts to the change from a traditional environment to a cloud environment. The report presented first the terminology intended to provide the fundamentals of the subject. Although, this discussion was to some extent extensive, it is justified by the usual confusion coming from the ambiguity of usage of some of the terms in the field and also by the abstractness of the subject.

The rest of this final section discusses and summarizes important aspects of the research process, summarizes the results and outlines potential paths for extending this study.

7.1 Discussion of research method

The design science research process proved to be a valuable choice of a research methodology. The principles, guidelines and prescriptions of the design science research process helped in ensuring a rigorous research and utility of the results. Despite the good fit of the research methodology, a number of weaknesses such as the need for extended Demonstration and Evaluation stages and the lack of external iterations of the study are recognized when the implementation of the process in this research is considered. These weaknesses were already mentioned in the respective sections and, on the positive end, options for their improvement were discussed which is at least some start for a building on the results of this research.

Four research questions were asked in the beginning of the report:

RQ1. How does the existing knowledge on architectural patterns relate to the cloud computing environment?

RQ2. Which characteristics of architectural patterns make them suitable for the cloud environment?

RQ3. How can architectural pattern evolution be documented effectively for usage in the practice?

These were addressed by constructing 4 artefacts:

1. a cloud applicability taxonomy of architectural patterns (CATAP);
2. a pattern-to-characteristics mapping;
3. a pattern form (CAPF);
4. a wiki, called APE Wiki available at http://patterns.freehosting.bg/mw.

These artefacts manage to demonstrate in a structured way the results of the research carried out during this thesis work. The artefacts would benefit, though, from external validation such as community engagement through the wiki or field studies revealing how exactly and to what extent these results help practitioners in the knowledge transfer.

7.2 Discussion of findings

An important finding is that the CATAP did not include any Evolution patterns which is a signal for the good maturity of the pattern field.

Another important result from the design of the artefacts is that the move to a cloud context does not "simply" result in a paradigm shift – there are new patterns identified which shows that not only the existing knowledge should be taken from a new perspective but also new knowledge is needed. This opens opportunities for exploring the field or even inventing new patterns which could serve the new environment better.

Another contribution of this thesis to the knowledge base is the patterns-to-characteristics mapping. The literature review did not show other studies which treat architectural patterns in the context of a structured analysis based on both the characteristics and the environment. The mapping, though, is rather general and would benefit from further iterations and further elaborating on documenting the relationships.

7.3 Options for future research

An option for extending the study regarding the CATAP is to make it a pattern language focused on solving a particular problem, building on the suggested taxonomy. As discussed in the literature review there exist methods and frameworks that could potentially use the taxonomy for achieving such a systematic approach, but this is out of the scope of this thesis.

Another option is to re-use the FPF and its relation to CAPF as an example and to apply the same pattern of structuring to other environments of potential interest e.g. embedded systems or mobile development. As shown in this research, concentrating on the special characteristics and effects a specific system environment has on the architectural patterns can reveal interesting and valuable information about the architectural patterns, whereas the structured way of presenting and abstracting the information through the FPF and an APF targeting an environment of interest could help in systemizing that information.

Apart from that, in the spirit of the gaining momentum work on semantic web, it could be interesting to see the artefacts of the research presented as an ontology. There are numerous relations between architectural patterns, patterns and characteristics, characteristics on their own which could be part of a pattern ontology and there already exists research about capturing architectural and pattern knowledge in ontologies (Henninger & Ashokkumar, 2006; Dietrich & Elgar, 2007; Zhao, et al., 2009). MediaWiki, in that regard, is also a good choice as a wiki engine as it has a Semantic MediaWiki extension (Krötzsch & Vrandecic, 2015).

Another way to build on the study presented here is to look at the patterns falling in the Oblivion group and searching for ways to make them more applicable and attractive for the cloud environment – each pattern is valuable on its own and potentially cloud engineering may benefit from having each and any of them at the disposal of cloud system architects.

References

@KenTamagawa, @c9katayama & @suz_lab, 2012. *CDP:Queuing Chain Pattern.* [Online]
Available at:
http://en.clouddesignpattern.org/index.php/CDP:Queuing_Chain_Pattern
[Accessed 25 Februari 2015].

Ahmad, A., Jamshidi, P. & Pahl, C., 2012. Pattern-driven Reuse in Architecture-centric Evolution for Service Software. In: *7th International Conference on Software Paradigm Trends ICSOFT'2012.* s.l.:s.n.

Alexander, C. et al., 1977. *A Pattern Language: Towns, Buildings, Construction.* New York: Oxford University Press.

AppHarbor, 2015. *AppHarbor - .NET Cloud Platform as a Service.* [Online]
Available at: https://appharbor.com/
[Accessed 28 April 2015].

Arnott, D. & Pervan, G., 2008. An assessment of DSS design science using the Hevner, March, Park and Ram guidelines . In: S. Gregor & D. Hart, eds. *Information Systems Foundations: The Role of Design Science.* Canberra, Australia: ANU E Press, pp. 255-284.

Avgeriou, P. & Zdun, U., 2005. Architectural Patterns Revisited: A Pattern Language. In: *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005).* Irsee: s.n., pp. 1-39.

Bachmann, F., Bass, L. & Nord, R., 2007. *Modifiability Tactics,* Pittsburgh: SEI, Carnegie Mellon University.

Bass, L., Clements, P. & Kazman, R., 2012. *Software Architecture in Practice.* 3rd ed. New Jersey: Addison-Wesley Professional.

Bernstein, P. et al., 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability,* Online: Microsoft Research.

Betts, D. et al., 2012. Reference 2: Introducing the Command Query Responsibility Segregation Pattern. In: *Exploring CQRS and Event Sourcing.* s.l.:Microsoft Corporation, pp. 223-235.

Buschmann, F., Henney, K. & Schmidt, D. C., 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing.* s.l.:John Wiley & Sons.

Buschmann, F., Henney, K. & Schmidt, D. C., 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages.* s.l.:John Wiley & Sons.

Buschmann, F. et al., 1996. *Pattern-Oriented Software Architecture Volume 1: A system of patterns.* s.l.:John Wiley & Sons.

Bykov, S. et al., 2011. Orleans: cloud computing for everyone. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing.* New York, NY, USA: ACM, pp. 16:1-16:14.

CIO Research Center, 2010. *SAAS, PAAS and IAAS – Making Cloud Computing Less Cloudy.* [Online]
Available at: http://cioresearchcenter.com/2010/12/107/
[Accessed 26 May 2014].

Clements, P. et al., 2010. *Documenting Software Architectures - Views and Beyond.* 2nd ed. Boston: Addison-Wesley.

Community, 2013. *AWS Cloud Design Patterns (CDP).* [Online]
Available at: http://en.clouddesignpattern.org/index.php/Main_Page
[Accessed 30 March 2015].

Community, 2015. *High Scalability.* [Online]
Available at: http://highscalability.com/
[Accessed 27 April 2015].

Corrêa, F., 2009. Actors in a new "highly parallel" world. In: *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010.* New York, NY, USA: ACM, pp. 21-24.

Côté, I., Heisel, M. & Wentzlaff, I., 2007. *Pattern-Based Evolution of Software Architectures.* Aranjuez, Spain, Springer Berlin Heidelberg, pp. 29-43.

Cunningham & Cunningham, Inc., 2005. *Portland Pattern Repository's Wiki: Pattern Forms.* [Online]
Available at: http://c2.com/cgi/wiki?PatternForms
[Accessed 1 May 2014].

Cunningham, W. & Beck, K., 1987. *Constructing Abstractions for Object-Oriented Applications,* s.l.: Computer Research Laboratory, Tektronix, Inc..

Dietrich, J. & Elgar, C., 2007. Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web,* 5(2), pp. 108-116.

Dong, J., Yang, S. & Sun, Y., 2007. A Classification of Design Pattern Evolutions. *Journal of Object Technology,* 6(10), pp. 95-109.

Dong, J., Yang, S. & Zhang, K., 2006. A Model Transformation Approach for Design Pattern Evolutions. In: *Proceedings of 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems.* Potsdam, Germany: IEEE, pp. 92-102.

Dong, J., Zhao, Y. & Sun, Y., 2010. Design pattern evolutions in QVT. *Software Quality Journal,* 18(2), pp. 269-297.

Dykstra, T., Anderson, R. & Wasson, M., 2014. *Building Real-World Cloud Apps with Windows Azure,* s.l.: Microsoft Corporation.

Erl, T., 2005. *Service-oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River: Prentice Hall PTR.

Erl, T. & Naserpour, A., 2015. *Cloud Computing Design Patterns.* s.l.:Prentice Hall/PearsonPTR.

Erl, T., Puttini, R. & Mahmood, Z., 2014. *Cloud Computing: Concepts, Technology & Architecture.* New Jersey: Prentice Hall/PearsonPTR.

Fehling, C. et al., 2011. *An Architectural Pattern Language of Cloud-based Applications.* Portland, Oregon, ACM, pp. 2:1-2:11.

Fehling, C. et al., 2014a. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Wien: Springer.

Fehling, C. et al., 2014b. Cloud Application Architecture Patterns. In: *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Wien: SpringerLink, pp. 151-238.

Fehling, C. et al., 2014c. *Cloud Computing Patterns.* [Online]
Available at: http://www.cloudcomputingpatterns.org/
[Accessed 9 May 2014].

Fowler, M., 2011. *CQRS.* [Online]
Available at: http://martinfowler.com/bliki/CQRS.html
[Accessed 10 March 2015].

Fowler, M. et al., 2002. *Patterns of Enterprise Application Architecture.* Boston: Addison Wesley.

Gabriel, R. P., 1996. *Patterns of Software: Tales from the Software Community.* New York: Oxford University Press.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.

Ghag, S. S. & Bandopadhyaya, R., 2013. Technical Strategies and Architectural Patterns for Migrating Legacy Systems to the Cloud. In: Z. Mahmood & S. Saeed, eds. *Software Engineering Frameworks for the Cloud Computing Paradigm.* London: Springer-Verlag, pp. 235-254.

Guha, R., 2013. Impact of Semantic Web and Cloud Computing Platform on Software Engineering. In: Z. Mahmood & S. Saeed, eds. *Software Engineering Frameworks for the Cloud Computing Paradigm.* London: Springer-Verlag, pp. 3-24.

Harrison, N., Avgeriou, P. & Zdun, U., 2007. Using Patterns to Capture Architectural Decisions. *IEEE Software,* 24(4), pp. 38-45.

Harrison, N., Avgeriou, P. & Zdun, U., 2010. On the Impact of Fault Tolerance Tactics on Architecture Patterns. In: *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems.* London: ACM, pp. 12-21.

Heesch, U. v., Avgeriou, P. & Hillard, R., 2012. A documentation framework for architecture decisions. *The Journal of Systems and Software,* 85(4), pp. 795-820.

Heesch, U. v., Hazavehi, S. M. & Avgeriou, P., 2011. Combining Architectural Patterns and Software Technologies in One Design Language. In: *Proceedings of the 16th European Pattern Languages of Programming (EuroPLoP 2011).* s.l.:s.n., pp. 1-18.

Henninger, S. & Ashokkumar, P., 2006. *An Ontology-Based Metamodel for Software,* University of Nebraska - Lincoln: Department of Computer Science and Engineering.

Heroku, Inc., 2015. *Heroku - Cloud Application Platform.* [Online]
Available at: https://www.heroku.com/
[Accessed 28 April 2015].

Hevner, A. & Chatterjee, S., 2010a. Introduction to Design Science Research. In: A. Hevner & S. Chatterjee, eds. *Design Research in Information Systems: Theory and Practice.* New York: Springer US, pp. 1-8.

Hevner, A. & Chatterjee, S., 2010b. Design Science Research in Information Systems. In: A. Hevner & S. Chatterjee, eds. *Design Research in Information Systems: Theory and Practice.* New York: Springer US, pp. 9-22.

Hevner, A., March, S., Park, J. & Ram, S., 2004. Design Science in Information Systems Research. *MIS Quarterly,* 28(1), pp. 75-105.

Hewitt, C., 2015. *Actor Model of Computation: Scalable Robust Information Systems,* New York: Cornell University.

Hohpe, G. & Woolf, B., 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston: Addison Wesley.

Hu, J. et al., 2013. Designing the Internet of Things for learning environmentally responsible behaviour. *Interactive Learning Environments,* 21(2), pp. 211-226.

ISO/IEC 25010:2011, 2011b. *Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models,* Geneva, Switzerland: International Organization for Standardization.

ISO/IEC IS 9126-1, 2001. *Software Engineering - Product Quality – Part 1: Quality Model,* Geneva, Switzerland: International Organization for Standarization.

ISO, 2011a. *ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.* [Online]
Available at: http://www.iso-architecture.org/42010/faq.html
[Accessed 01 May 2014].

Kazman, R., Klein, M. & Clements, P., 2000. *ATAM: Method for Architecture Evaluation,* Pittsburgh: SEI, Carnegie Mellon University.

Kim, D.-K., 2013. Design pattern based model transformation with tool support. *Software: Practice and Experience.*

Kircher, M. & Jain, P., 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management.* s.l.:John Wiley & Sons.

Kobayashi, T. & Saeki, M., 1999. Software Development Based on Software Pattern Evolution. In: *Proceedings of Sixth Asia Pacific Software Engineering Conference, 1999 (APSEC'99).* Washington, USA: IEEE, pp. 18-25.

Krishna, R. & Jayakrishnan, R., 2013. Impact of Cloud Services on Software Development Life Cycle. In: M. Zaigham & S. Saeed, eds. *Software Engineering Frameworks for the Cloud Computing Paradigm.* London: Springer, pp. 79-99.

Krötzsch, M. & Vrandecic, D., 2015. *SMW: Semantic Media Wiki.* [Online]
Available at: https://semantic-mediawiki.org/wiki/Semantic_MediaWiki
[Accessed 12 May 2015].

Kruchten, P., 1995. The 4+1 view model of architecture. *IEEE Software,* 12(6), pp. 42-50.

Kuhn, T. S., 1962. *The Structure of Scientific Revolutions.* Chicago: University of Chicago Press.

Lehman, M. et al., 1997. Metrics and Laws of Software Evolution - the Nineties View. In: *Proceedings of the Fourth International Software Metrics Symposium.* Albuquerque, USA: IEEE, pp. 20-32.

Marinescu, D., 2013. *Cloud computing theory and practice.* Boston: Morgan Kaufmann.

McKendrick, J., 2011. *Cloud Computing's Vendor Lock-In Problem: Why the Industry is Taking a Step Backward.* [Online]
Available at:
http://www.forbes.com/sites/joemckendrick/2011/11/20/cloud-computings-vendor-lock-in-problem-why-the-industry-is-taking-a-step-backwards/
[Accessed 23 March 2015].

Mezirow, J., 1997. Transformative learning: Theory to practice. *New directions for adult and continuing education,* 1997(74), pp. 5-12.

Microsoft Corporation, 2014. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/dn568099.aspx
[Accessed 30 March 2015].

Microsoft Corporation, 2014. *Command and Query Responsibility Segregation (CQRS) Pattern.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/dn568103.aspx
[Accessed 12 March 2015].

Microsoft Corporation, 2015. *Microsoft Azure.* [Online]
Available at: http://www.azure.microsoft.com
[Accessed 28 April 2015].

Miguel, J., Mauricio, D. & Rodriguez, G., 2014. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications,* 5(6), pp. 1-24.

Moyer, C., 2011. *Building Applications in the Cloud: Concepts, Patterns, and Projects.* Boston: Addison-Wesley Professional.

Netflix, 2015. *The Netflix Tech Blog.* [Online]
Available at: http://techblog.netflix.com/
[Accessed 27 March 2015].

NIST: Mell, P. & Grance, T., 2011. *The NIST Definition of Cloud Computing.* [Online]
Available at: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
[Accessed 7 March 2014].

Nunamaker, J., Chen, M. & Purdin, T., 1990-1991. Systems Development in Information Systems Research. *Journal of Management Information Systems,* 7(3), pp. 89-106.

Ortega-Arjona, J. L., 2010. *Patterns for Parallel Software Design.* Chichester, UK: John Wiley & Sons.

Pattern Community, 2007. *Beck Form.* [Online]
Available at: http://c2.com/cgi/wiki?BeckForm
[Accessed 20 May 2014].

Pattern Community, 2010. *Wiki Choicetree.* [Online]
Available at: http://c2.com/cgi/wiki?WikiChoicetree
[Accessed 15 May 2014].

Pattern Community, 2011. *Compact Form.* [Online]
Available at: http://c2.com/cgi/wiki?CompactForm
[Accessed 20 May 2014].

Pattern community, 2013. *Pattern Index.* [Online]
Available at: http://c2.com/cgi/wiki?PatternIndex
[Accessed 5 July 2015].

Pattern Community, 2014. *Wiki Engines.* [Online]
Available at: http://c2.com/cgi/wiki?WikiEngines
[Accessed 15 May 2014].

Peffers, K., Tuunanen, T., Rothenberger, M. & Chaterjee, S., 2007-8. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems,* 24(3), pp. 45-77.

Pramod, N., Muppala, A. K. & Srinivasa, K., 2013. Limitations and Challenges in Cloud-Based Applications Development. In: M. Zaigham & S. Saeed, eds. *Software Engineering Frameworks for the Cloud Computing Paradigm.* London: Springer, pp. 55-75.

Ram, J. & Rajasree, M., 2003. Enabling Design Evolution in Software through Pattern-Oriented Approach. *Object-Oriented Information Systems,* pp. 179-190.

Rozanski, N. & Woods, E., 2011. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition.* 2nd ed. Boston: Addison-Wesley.

Schmidt, D. C., Stal, M., Rohnert, H. & Buschmann, F., 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* s.l.:John Wiley & Sons.

Scott, J. & Kazman, R., 2009. *Realizing and Refining Architectural Tactics: Availability,* Pittsburgh: SEI, Carnegie Mellon University.

SEI, 2014. *Architecture Tradeoff Analysis Method.* [Online]
Available at:
http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm
[Accessed 16 July 2014].

SEI, 2014. *Community Software Architecture Definitions.* [Online]
Available at:
http://www.sei.cmu.edu/architecture/start/glossary/community.cfm
[Accessed 25 April 2014].

Strauch, S. et al., 2013. Using Patterns to Move the Application Data Layer to the Cloud. In: *PATTERNS 2013 : The FIfth International Conferences on Pervasive Patterns and Applications.* Valencia, Spain: International Academy, Research and Industry Association (IARIA), pp. 26-33.

Tokuda, L. & Batory, D., 1995. Automated Software Evolution via Design Pattern Transformations. In: *Proceedings of the 3rd International Symposium on Applied Corporate Computing.* Monterrey, Mexico: s.n.

Varia, J., 2011. *Architecting for The Cloud: Best Practices.* [Online]
Available at:
http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
[Accessed 3 May 2014].

Vogel, O., Arnold, I., Chughtai, A. & Kehrer, T., 2011. *Software Architecture: A Comprehensive Framework and Guide for Practitioners.* Berlin Heidelberg: Springer.

Wikipedia, 2014a. *Category:Systems theory.* [Online]
Available at: https://en.wikipedia.org/wiki/Category:Systems_theory
[Accessed 1 April 2015].

Wikipedia, 2014b. *Template:Infobox.* [Online]
Available at: http://en.wikipedia.org/wiki/Template:Infobox
[Accessed 11 May 2015].

Wikipedia, 2015. *Service-level agreement.* [Online]
Available at: https://en.wikipedia.org/wiki/Service-level_agreement
[Accessed 10 June 2015].

Wilder, B., 2012. *Cloud Architecture Patterns,* Safari Books Online: O'Reilly Media, Inc..

Williamson, K., 2002. *Research methods for students, academics and professionals: information management and systems.* 2nd ed. Wagga Wagga, N.S.W.: Centre for Information Studies, Charles Sturt University .

Zdun, U., Avgeriou, P., Hentrich, C. & Dustdar, S., 2008. Architecting as Decision Making with Patterns and Primitives. In: *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge.* Leipzig, Germany: ACM, pp. 11-18.

Zhang, Q., Cheng, L. & Boutaba, R., 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications,* I(1), pp. 7-18.

Zhao, C., Kong, J., Dong, J. & Zhang, K., 2007. Pattern-based design evolution using graph transformation. *Journal of Visual Languages and Computing,* 18(4), pp. 378-398.

Zhao, Y., Dong, J. & Peng, T., 2009. Ontology Classification for Semantic-Web-Based Software Engineering. *IEEE Transactions on Services Computing,* 2(4), pp. 303-317.

Terms and Definitions

Adaptation view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It looks at systems as consisting of invariable and adaptable components
Architectural design principle	"fundamental statement of belief, approach, or intent that guides the definition of your architecture" (Rozanski & Woods, 2011)
Architectural pattern	An architectural pattern describes "a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them" (Buschmann, et al., 1996). It affects the system as a whole.
Architectural Patterns' Evolution (APE) Wiki	Artefact 4 in this report. A publicly available wiki system intended to share the results of this thesis research with the practitioners' community
Architectural view	"a representation of a system from the perspective of a related set of concerns" (Avgeriou & Zdun, 2005, p. 4)
Broad network access	An essential cloud computing characteristic. "Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platform" (NIST: Mell & Grance, 2011, p. 2)
Cloud (computing) environment	The system environment associated with cloud computing. This environment has special characteristics and imposes specific design, development, deployment and distribution models.
Cloud abbreviated pattern form (CAPF)	Artefact 4 in this report. A subset of the Full pattern form (FPF) which concentrates on the cloud computing environment considerations only.
Cloud applicability categories	These are 4 groups defining categories of architectural pattern depending on whether and how they can be used in a cloud computing environment. The cloud applicability categories are: Transfer (applicable without changes), Evolution (applicable through significant adaptation), Innovation (completely new patterns that

	have emerged for the cloud specifically), and Oblivion (applicable in limited scenarios or not recommended).
Cloud applicability taxonomy of architectural patterns (CATAP)	Artefact 1 in this report. A taxonomy of architectural patterns classifying architectural patterns into cloud applicability categories
Cloud application architectural pattern (CAAP)	An architectural pattern suitable for building cloud-native applications
Cloud architectural pattern	An architectural pattern considered in a cloud context
Cloud computing	"A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." (NIST: Mell & Grance, 2011)
Cloud computing consumer	A person or an organization which rents cloud computing resources from a cloud computing vendor.
Cloud computing vendor	A person or an organization which owns, defines, maintains, and rents out cloud computing resources to cloud computing consumers.
Cloud infrastructural architectural pattern (CIAP)	An architectural pattern that deals with the specific software architectural concerns when building cloud infrastructure
Cloud infrastructure	"the collection of hardware and software that enables the five essential characteristics of cloud computing" (NIST: Mell & Grance, 2011, p. 2)
Cloud-native application	A system deployed to the cloud, utilizing the specific cloud characteristics
Community cloud	A cloud deployment model. "The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns […] It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises" (NIST: Mell & Grance, 2011, p. 2)

Component Interaction view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It focuses on communication through messages which also preserves the components' autonomy
Data Flow view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It looks into how a system's components can sequentially process and transform data streams
Data-centred view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It is concerned with utilizing a central repository of data accessible by multiple components
Design pattern	A design pattern provides "a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context" (Buschmann, et al., 1996). It affects only a part of a system and is considered during implementation.
Design science research	"A research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artifacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem" (Hevner & Chatterjee, 2010a)
Design science research methodology (DSRM)	A methodology designed by Peffers et al. (2007-8) in order to provide a systematic way of applying design science research in practice
Distribution view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It looks into systems as distributed in a networked environment
Full pattern form (FPF)	A pattern form designed during the design of Artefact 3, the Cloud abbreviated pattern form (CAPF). It is based on standard widely-used pattern forms and extends them with additional information about the cloud context
Hybrid cloud	A cloud deployment model. "The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or

	proprietary technology that enables data and application portability" (NIST: Mell & Grance, 2011, p. 2)
Idiom	A language-level pattern describing "how to implement particular aspects of components or the relationships between them using the features of the given language" (Buschmann, et al., 1996)
Infrastructure as a Service (IaaS)	A cloud service model. Allows the cloud computing consumer to "provision processing, storage, networks, and other fundamental computing resources" (NIST: Mell & Grance, 2011, p. 2)
Language Extension view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It looks at the system as a part of the external world which it communicates with through a form of abstraction
Layered view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It looks into a system in terms of how it can be structured as interacting parts
Measured service	An essential cloud computing characteristic. "Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service […]. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service" (NIST: Mell & Grance, 2011, p. 2)
National Institute of Standards and Technology (NIST)	The US federal technology agency that works with industry to develop and apply technology, measurements, and standards
On-demand self-service	An essential cloud computing characteristic. "A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider." (NIST: Mell & Grance, 2011, p. 2)
On-premises deployment	Deployment of a system on owned hardware. Alternative name is in-house deployment.
Pattern	A pattern according to this definition "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this

	solution a million times over, without ever doing it the same way twice" (Alexander, et al., 1977)
Pattern evolution	The changes that a pattern as a description or a template of a solution goes through because of a change in the context, the problems solved or any other external to the pattern factor which the pattern depends on
Pattern form	A predefined structured way of presenting the information about patterns
Pattern instance	A concrete implementation of a pattern
Pattern language	"A network of interrelated patterns that defines a process for resolving software development problems systematically" (Buschmann, et al., 2007b)
Pattern-to-characteristics mapping	Artefact 2 in this report. It represents a mapping between architectural patterns and the system characteristics they affect
Platform as a Service (PaaS)	A cloud service model. Gives cloud computing consumers the ability to deploy to the cloud infrastructure and build their own applications on top of a mostly predefined infrastructure (NIST: Mell & Grance, 2011, p. 2)
Private cloud	A cloud deployment model. "The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises." (NIST: Mell & Grance, 2011, p. 2)
Public cloud	A cloud deployment model. "The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider" (NIST: Mell & Grance, 2011, p. 2)
Rapid elasticity	An essential cloud computing characteristic. "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand" (NIST: Mell & Grance, 2011, p. 2)

Resource pooling	An essential cloud computing characteristic. "The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand" (NIST: Mell & Grance, 2011, p. 2)
Service-level Agreement (SLA)	A part of a service contract where a service is formally defined. Particular aspects of the service - scope, quality, responsibilities - are agreed between the service provider and the service user (Wikipedia, 2015)
Software architecture	"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (ISO, 2011a) "the set of design decisions which, if made incorrectly, may cause your project to be cancelled" (Woods, 2014)
Software as a Service (SaaS)	A cloud service model. Provides cloud computing consumers with the ability to consume a cloud-running service/application with a very limited possibilities for configuration (NIST: Mell & Grance, 2011, p. 2)
Software Engineering Institute (SEI)	A US federally funded research and development centre which has initiatives aimed at improving organizations' software engineering capabilities e.g. management and engineering practices, security.
Software pattern	A cumulative name for a design or an architectural pattern
Traditional architectural pattern	An architectural pattern considered in a context different from cloud e.g. on-premises solutions.
Traditional computing model	The traditional computing model encompasses 2 options: (1) to host a system as on-premises deployment, or (2) to use a hosting provider which provides a predefined amount of hardware and software resources either dedicated, or shared with other consumers.
User Interaction view	An architectural view used in Avgeriou & Zdun's architectural pattern language (2005). It is concerned with "the runtime structure of components that offer a user interface" (Avgeriou & Zdun, 2005)

Search Terms

Architectural view
Adaptation view 34, 62, 72, 73, 74
Component Interaction view 35, 36, 64, 73, 74
Data Flow view 64
Data-centred view 34, 64, 72
Distribution view 31, 36, 65, 73, 74
Language Extension view 29, 65
Layered view 65, 72
User Interaction view 34, 36, 68, 73, 74
Artefacts
APE Wiki ..... iii, vi, 10, 20, 22, 23, 33, 46, 47, 48, 49, 50, 53, 54, 55, 63
CATAP ..... ii, vi, 10, 21, 22, 24, 25, 28, 29, 33, 35, 42, 43, 48, 53, 55, 56, 63, 71, 72
Pattern form ..... vi, 22, 35, 36, 37, 44, 45, 46, 49, 56, 63, 65, 71, 81
Patterns-to-characteristics mapping 22, 35, 37, 39, 43, 53, 67, 71, 80
Characteristics
Analysability 79
Availability ..... vi, 9, 11, 12, 14, 17, 32, 38, 39, 40, 41, 74, 77, 82
Fault tolerance ..... vi, 40, 41, 77
Interoperability 76
Location transparency ..... vi, 2, 35, 36, 40, 41, 73
Modifiability 79
Modularity 2, 4, 11, 38, 41, 73
Multi-tenancy 2, 36, 76
Performance 36, 75, 82
Portability 36, 41, 79
Recoverability ..... vi, 39, 40, 41, 82
Reusability 78
Scalability ..... vi, 2, 3, 27, 31, 33, 34, 35, 38, 40, 41, 67, 72, 73, 75, 81, 82
Security 36, 78
Testability 79
Cloud applicability category 48
Evolution ..... vi, 10, 21, 22, 24, 33, 56, 63
Innovation 10, 21, 24, 28, 39, 40, 63, 73
Oblivion 10, 8, 21, 24, 32, 34, 35, 40, 41, 56, 63, 72, 73, 74
Transfer ..... iii, 10, 21, 24, 33, 35, 39, 40, 41, 63, 72, 73, 74
Cloud computing ..... vi, 8, 2, 4, 11, 31, 63, 64
Deployment model 4, 31
Essential characteristics 2, 4, 27, 30, 31, 38, 74, 75
Infrastructure 3, 4, 7, 11, 30, 31, 32, 64, 65, 67, 77
Service model ..... vi, 3, 11, 30, 65, 67, 68
Design science 15, 16, 17, 65
Design science artefact 20, 21, 22, 24, 28, 35, 36, 37, 44, 46, 49, 63, 65, 67, 71, 81
Design science research methodology ..... vi, 16, 17, 18, 19, 65
Full pattern form ..... vi, 44, 45, 46, 49, 56, 63, 65

Ontology 56
Pattern evolution 7, 14, 66
Pattern form 42, 66
Pattern language 10, 66
Patterns
Architectural pattern 7, 8, 6, 7, 12, 13, 42, 62, 68
ACTIVE REPOSITORY 34, 72
BATCH SEQUENTIAL 72
BLACKBOARD 72
BROKER 30, 31, 32, 39, 72, 74
CLIENT-SERVER 74, 82
CQRS ..... vi, 29, 30, 32, 33, 39, 73
EXPLICIT INVOCATION 74
IMPLICIT INVOCATION 73
INDIRECTION LAYER 30, 31, 32, 39, 72, 82
INTERCEPTOR 72, 73, 74
LAYERS 27, 29, 30, 31, 32, 38, 39, 46, 71, 72, 81, 82
MESSAGE QUEUING 27, 32, 39, 73, 74
MICROKERNEL 73, 82
PEER-TO-PEER 74
PIPES AND FILTERS 8, 27, 72, 82
PUBLISH-SUBSCRIBE 73
REFLECTION 72, 74
REMOTE PROCEDURE CALLS 74
RULE-BASED SYSTEM 29
SHARED REPOSITORY 34, 72, 73, 82
Cloud considerations
Cloud application architectural pattern ..... i, vi, 7, 9, 18, 63
Cloud architectural pattern 7
Cloud infrastructure architectural pattern ..... vi, 7, 28, 64
Design pattern 7, 64, 68
Pattern instance 6, 13, 14
Semantic web 56
Software architecture ..... i, ii, vi, 7, 8, 10, 3, 4, 5, 6, 12, 13, 14, 27, 35, 36

Appendices

Appendix 1: Cloud applicability taxonomy of architectural patterns (CATAP)

Appendix 2: Relevance matrix for the characteristics defined in

Appendix 3: Patterns-to-characteristics mapping

Appendix 4: LAYERS pattern documented using Artefact 3, a pattern form

Appendix 1: Cloud applicability taxonomy of architectural patterns (CATAP)

Pattern (Avgeriou & Zdun *(2005)* view)	Cloud category	Comment
LAYERS (Layered view)	Transfer	Not just transfers but becomes almost a requirement for an application to be considered a true cloud application
INDIRECTION LAYER (Layered view)	Transfer	Grows in importance especially for hybrid systems and systems moving to cloud in which cases may often function as a BROKER. Called Data Access component in (Fehling, et al., 2014c, pp. 188-193); Data Abstractor is a variant (pp. 194-196)
BATCH SEQUENTIAL (Data-flow view)	Transfer	Grows in importance and more applicable. The good scalability and parallelism are a good match for cloud environments. By using message-oriented middleware and different message delivery strategies which are available in cloud (Fehling, et al., 2014b, pp. 136-150; @KenTamagawa, et al., 2012), some of its drawbacks can be overcome.
PIPES AND FILTERS (Data-flow view)	Transfer	Same as BATCH SEQUENTIAL
SHARED REPOSITORY (Data-centred view)	Transfer	It has many possible use cases. Still a consideration it can be a bottleneck in the system, but this is not a change compared to traditional environments. Fehling, et al. (2014c, p. 182) even discuss it in a combination with PIPES AND FILTERS.
ACTIVE REPOSITORY (Data-centred view)	Transfer	Based on SHARED REPOSITORY. The pattern has its advantages and popular scenarios such as push notifications are based on it, thus applicable for cloud.
BLACKBOARD (Data-centred view)	Transfer	Hard to decide in between Transfer and Oblivion; Transfer for the applicability in the emerging Internet of Things as a public storage for devices as knowledge sources (Hu, et al., 2013).
INTERCEPTOR (Adaptation view)	Transfer	The pattern is part of the Transfer group as it provides an important aspect of a cloud scenario – updating the system. As a cloud system is usually expected to operate 24/7 downtime

		for updates is not tolerated and interceptor can help with a dynamic, gradual and controlled release of new services to the customers. It can be combined with REFLECTION
MODEL-VIEW-CONTROLLER (User Interaction view)	Transfer	The three patterns in the User Interaction share the same considerations. The user interaction part of a system is usually a loosely-coupled component (usually a thin client containing only user interface) or even a semi-autonomous sub-system having its own architecture (thick client, containing user interface, application logic and even data model). Different technological solutions (programming frameworks and libraries) have been designed to address the considerations of user interaction in full which introduces a two-way dependency of architectural and implementation decisions – team knowledge and competencies may have bigger impact on architectural decisions than purely architectural concerns; such frameworks also often predispose for a specific way of communication thus a specific Component interaction view pattern. Considering the architectural part, an important design decision is how the application logic and interface logic are separated and what is communicated between them – all these three patterns support different scenarios and are applicable depending on the case.
PRESENTATION-ABSTRACTION-CONTROL (User Interaction view)	Transfer	
C2 (User Interaction view)	Transfer	
IMPLICIT INVOCATION (Component Interaction view)	Transfer	Although the pattern transfers, there are some limitations imposed on it – synchronous invocations are hardly relevant – blocking resources is costly and fault tolerance is reduced.
PUBLISH-SUBSCRIBE (Component Interaction view)	Transfer	The combination with SHARED REPOSITORY and the loose coupling the pattern promotes are valuable in the cloud environment. Popular scenarios such as push notifications are based on this pattern.
MESSAGE QUEUING (Distribution view)	Transfer	Almost a standard for cloud applications as it supports location-agnostic and fault tolerant systems
COMMAND QUERY RESPONSIBILITY SEGREGATION	Innovation	Completely new; enables using elasticity for providing better performance
MICROKERNEL (Adaptation view)	Oblivion	Although the ideas of MICROKERNEL could be used in a cloud scenario, the INTERCEPTOR pattern is less complex and emphasizes on building upon a framework which

		usually is the case for service-oriented systems thus applicable for many cloud systems, rather than a system kernel.
REFLECTION (Adaptation view)	Oblivion	Although the pattern could potentially be used in cloud e.g. in a combination with an INTERCEPTOR or in the architecture of sub-systems, it may be harder to control, analyse and troubleshoot a cloud system build with it. Still, it could be considered on the border of Transfer and Oblivion, but in this study it is considered to slightly better fit in Oblivion.
EXPLICIT INVOCATION (Component Interaction view)	Oblivion	Being in the Oblivion group for this pattern does not mean it is not used – if a system has both a cloud and a non-cloud parts e.g. mobile applications using cloud-based services, these communicate through EXPLICIT INVOCATION, but this is a consequence of the way Internet works. In that case the communication is usually asynchronous as network latency, reliability and even availability is highly unpredictable, which affects user experience (see the patterns from the User Interaction view for the important of user experience). Within the cloud parts of a system, tying components tightly and especially dependent on location is undesirable due to resource pooling and often lack of node-based availability (Fehling, et al., 2014c, pp. 95-97) guarantee.
CLIENT-SERVER (Component Interaction view)	Oblivion	A variant of the EXPLICIT INVOCATION pattern (see its comments, the mobile application in the example acts as a CLIENT and the cloud-part acts as a SERVER). If the location of the components is not hard-wired (Avgeriou & Zdun, 2005, p. 31) it has some potential for usage, but this is still not enough ground to be in Transfer. The CLIENT-SERVER pattern, though, poses an important concern to think about – what is part of the CLIENT and what is part of the SERVER, how the communication between them happens. This is also covered in the comments about the patterns in the User Interaction view.
PEER-TO-PEER (Component Interaction view)	Oblivion	A variant of the EXPLICIT INVOCATION. Although possible to use and in sync with the dynamic nature of cloud resources, it can be too complex and inefficient.
BROKER (Distribution view)	Oblivion	Still possible to implement but MESSAGE QUEUING is preferred.

REMOTE PROCEDURE CALLS (Distribution view)	Oblivion	Tying components together is undesirable in a cloud environment as often there is no node-based availability (Fehling, et al., 2014c, pp. 95-97)

Appendix 2: Relevance matrix for the characteristics defined in (ISO/IEC 25010:2011, 2011b)

Note: The characteristics in bold are the main eight characteristics in ISO/IEC 25010:2011, whereas the characteristics beneath them are their related subcharacteristics (ISO/IEC 25010:2011, 2011b).

(Sub)Characteristic	Description	Comments
Functional suitability	**degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions**	**The characteristic is important for a product and defines user acceptance, but it is not relevant for the research presented here**
Functional completeness	degree to which the set of functions covers all the specified tasks and user objectives	See the parent characteristic
Functional correctness	degree to which a product or system provides the correct results with the needed degree of precision	See the parent characteristic
Functional appropriateness	degree to which the functions facilitate the accomplishment of specified tasks and objectives	See the parent characteristic
Performance efficiency	**performance relative to the amount of resources used under stated conditions**	This characteristic is discussed through its subcharacteristics.
Time behaviour	degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements	These two characteristics are merged as Performance in the discussions in this paper as the effect of these on the level of pattern abstraction is hard to assess alone - they are rather dependent on particular system implementation. The view on Resource utilization is in fact one of the reasons to have the notion of cloud-native applications.
Resource utilization	degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements	
Capacity	degree to which the maximum limits of a product or system parameter meet requirements	In this paper, Capacity is replaced with Scalability as it can be qualified for patterns. For cloud-native applications Elasticity (as in rapid elasticity) is the more proper term as it denotes dynamic scalability.

Compatibility	**degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment**	This characteristic is discussed through its subcharacteristics
Co-existence	degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product	Usually cloud consumers share resources with other consumers on infrastructural level - usually the term multi-tenancy is used. Multi-tenancy is an important concept for cloud applications as well - system customization needs to be controlled on a logical system level, rather than on a deployment level as the support story for cloud consumers providing cloud systems to their users is rather complex and with high costs.
Interoperability	degree to which two or more systems, products or components can exchange information and use the information that has been exchanged	A characteristic of interest
Usability	**degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use**	This characteristic overall, although as tendency shows is very important for user adoption and product success is not strongly influenced by patterns.
Appropriateness recognizability	degree to which users can recognize whether a product or system is appropriate for their needs	Not applicable, see the Usability parent characteristic
Learnability	degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use	Not applicable, see the Usability parent characteristic
Operability	degree to which a product or system has attributes that make it easy to operate and control	Not applicable, see the Usability parent characteristic

User error protection	degree to which a system protects users against making errors	Not applicable, see the Usability parent characteristic
User interface aesthetics	degree to which a user interface enables pleasing and satisfying interaction for the user	Not applicable, see the Usability parent characteristic
Accessibility	degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use	Not applicable, see the Usability parent characteristic
Reliability	**degree to which a system, product or component performs specified functions under specified conditions for a specified period of time**	This characteristic is discussed through its subcharacteristics
Maturity	degree to which a system, product or component meets needs for reliability under normal operation	Maturity is hardly influenced by a specific pattern - patterns are solutions tested in time and are mature. Maturity is rather determined by the architectural decisions
Availability	degree to which a system, product or component is operational and accessible when required for use	Availability is partially dependent on the cloud infrastructure infrastructure, but cloud vendors provide better and better availability guarantees, thus the importance of ensuring availability on a system level grows in importance.
Fault tolerance	degree to which a system, product or component operates as intended despite the presence of hardware or software faults	Often cited by practitioners as very important for cloud systems; this is understandable as systems operate 24/7 and it is harder for administrators to change a system for a particular tenant independently.
Recoverability	degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system	Similar to Fault tolerance, the system should preferrably be able to recover automatically.

Security	**degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization**	Security's sucharacteristics are hardly related to the choice of a system pattern, but are rather dependent on technology and implementation. On the other hand, system architecture also has some impact on security, considering e.g. potential attack areas.
Confidentiality	degree to which a product or system ensures that data are accessible only to those authorized to have access	Not applicable, see the Security parent characteristic
Integrity	degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data	Not applicable, see the Security parent characteristic
Non-repudiation	degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later	Not applicable, see the Security parent characteristic
Accountability	degree to which the actions of an entity can be traced uniquely to the entity	Not applicable, see the Security parent characteristic
Authenticity	degree to which the identity of a subject or resource can be proved to be the one claimed	Not applicable, see the Security parent characteristic
Maintainability	**degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers**	This characteristic is discussed through its subcharacteristics
Modularity	degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components	Modularity is loose-coupling on a system level; as discussed this is an essential characteristic for distributed system such as cloud-native applications
Reusability	degree to which an asset can be used in more than one system, or in building other assets	A characteristic of interest

Analysability	degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified	Complementary to Modularity, Analysability is important for cloud-native systems as they have complex distribution and localizing a specific problem or change impact and considering the scenarios for coping with that should be as easy as possible.
Modifiability	degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality	A characteristic of interest
Testability	degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met	A characteristic of interest
Portability	**degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another**	Based on the discussions for its subcharacteristics, Portability will be discussed in terms of deployability and extent to which a system can potentially switch cloud providers.
Adaptability	degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments	The cloud environment is homogeneous and levels of abstraction limit the need for adaptation. Adaptability is a concern if switching between cloud vendors but this can be problematic as there is no standardization currently (McKendrick, 2011; Pramod, et al., 2013).
Installability	degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment	Installability in the case of cloud can be replaced with Deployability, which is an important story for releasing a cloud-native system, but is not dependent on patterns.
Replaceability	degree to which a product can replace another specified software product for the same purpose in the same environment	Not applicable for cloud applications.

Appendix 3: Patterns-to-characteristics mapping

Pattern	Cloud category	Performance	Elasticity (Scalability)	Multi-tenancy	Interoperability	Availability	Fault-tolerance	Recoverability	Modularity	Reusability	Analysability	Modifiability	Testability	Portability	Security	Location transparency
LAYERS	Transfer	−	▲	−	−	▲	−	−	−	−	−	−	−	−	−	−
INDIRECTION LAYER	Transfer	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
BATCH SEQUENTIAL	Transfer	▲	▲	−	−	▲	−	▲	−	−	−	−	−	−	−	▲
PIPES AND FILTERS	Transfer	▲	▲	−	−	▲	−	▲	−	−	−	−	−	−	−	▲
SHARED REPOSITORY	Transfer	▲	−	−	−	▲	−	▲	−	−	−	−	−	▼	−	−
ACTIVE REPOSITORY	Transfer	▲	−	−	−	▲	−	▲	−	−	−	−	−	▼	−	−
BLACKBOARD	Transfer	−	−	−	−	▲	−	▲	−	−	−	−	−	−	−	−
INTERCEPTOR	Transfer	−	▲	−	−	−	−	−	−	−	−	−	−	−	−	−
IMPLICIT INVOCATION	Transfer	−	▲	−	−	▲	▲	−	−	−	−	−	−	−	−	−
PUBLISH-SUBSCRIBE	Transfer	−	−	−	−	−	▲	−	−	−	−	−	−	−	−	−
MESSAGE QUEUING	Transfer	−	▲	−	−	▲	▲	▲	−	−	−	−	−	▼	−	−
CQRS	Innovation	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
MICROKERNEL	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	▼	−	−
REFLECTION	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
EXPLICIT INVOCATION	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
CLIENT-SERVER	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
PEER-TO-PEER	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
BROKER	Oblivion	−	−	−	−	−	−	−	−	−	−	−	−	−	−	−
REMOTE PROCEDURE CALLS	Oblivion	▼	−	−	−	−	−	−	−	−	−	−	▼	−	−	−

LEGEND	
Traditional environment	Cloud Environment
support	▲ enhance
neutral	− no change
hinders	▼ diminish

Appendix 4: LAYERS pattern documented using Artefact 3, a pattern form

Name: LAYERS (cloud) (Avgeriou & Zdun, 2005)

Context

The cloud environment is largely distributed and benefits from component decoupling. This means that in general the LAYERS pattern is a very potent choice for most cloud systems.

Problem

A system is being designed with functionalities which require both low- and high-level functionalities to be implemented in order to operate, because the environment on which the system is running does not provide services to make the high-level tasks easier to implement easily. Typically, this means that the communication flows from the higher level of abstraction to the lower level as a request and goes back to the higher level as a response.

In a cloud environment often a system should use a number of services provided by the cloud vendor. These services often have alternatives and also they evolve on their own - these dependencies need to be isolated so that changes in these external services would not affect the system operation.

Also, cloud environments in general enforces a layered thinking as the cloud platform services are decomposed into logical entities and their integration naturally happens in a layered manner.

The LAYERS pattern also helps in achieving elasticity, because the different layers could be scaled separately. Scalability is a common problem for systems built for cloud as they usually need to operate on a global big scale and the option to use a virtually unlimited amount of resources is very attractive and needed.

Solution

The LAYERS pattern is very simple on a conceptual level – decompose the application into logical layers, each containing components on the same abstraction level and allow, in the strict case, only adjacent layers to communicate with each other. That way, layers from lower-level provide services to higher-level layers.

Microsoft Azure (Microsoft Corporation, 2015), Heroku (Heroku, Inc., 2015), AppHarbor (AppHarbor, 2015) provide the notion of internet-connected and internet-disconnected resources (called web and worker roles, web and worker dynos, web and background workers respectively) – the former are connected to the Internet and serve as a web server processing web requests, whereas the latter are intended, for example, for intensive, potentially asynchronous business processing and running services in the background like an application server. This may help implementing layering in terms of cohesion - components serving web requests are separated from the operational components - and security - Internet-connected resources are more vulnerable to attacks.

Considering also that every cloud vendor has a number of different data services e.g. relational and NoSQL databases, in-memory databases, replication and backup,

constructing a data layer which can abstract data-related tasks away from the rest of the system can be rather beneficial. Considering systems which may not be very processing intensive but serve a lot of end users, separating a presentational layer on internet-connected resources can also be largely beneficial, because of the potential for scaling.

Consequences

Positives

Essentially, cloud systems using LAYERS benefit most from the support for scalability as each layer can be scaled horizontally independently. As a side effect, enhancing the scalability also enhances the availability of the system overall as the operational scaled instances of components can act as failover instances in case of a failure.

Negatives

No significant changes to the traditional pattern.

Performance concerns may be slightly softened because of the better scalability.

On the other hand, recoverability could be slightly harder because of the dynamic system environment and the limited control over it from the system maintainer.

Other considerations

No significant changes to the traditional pattern.

The effect on the cost can be slightly softened as often cloud resources are cheaper because of economy of scale.

Relations to other patterns

Two adjacent LAYERS can be considered as a CLIENT-SERVER pair, the higher layer being the client and the lower layer being the server. (Avgeriou & Zdun, 2005)

Similarly the PRESENTATION-ABSTRACTION-CONTROL pattern can also be thought in terms of LAYERS (Buschmann, et al., 1996).

A MICROKERNEL is also a layered architecture with three LAYERS: external servers, the microkernel, and internal servers.

An INDIRECTION LAYER pattern can be thought of as a two- or three-layered system, where one of the layers abstracts the layer beneath it.

The patterns PIPES AND FILTERS and SHARED REPOSITORY are somewhat opposite to the LAYERS pattern as all components in them should be on the same level of abstraction. On the hand, LAYERS can successfully be internally in these two patterns (Avgeriou & Zdun, 2005)