Juho Kettunen

Maintainability in cloud-native architecture

Master's Thesis in Information Technology

March 23, 2024

University of Jyväskylä

Faculty of Information Technology

Author: Juho Kettunen

Contact information: juho.kettunen@student.jyu.fi

Supervisor: Oleksiy Khriyenko

Title: Maintainability in cloud-native architecture

Työn nimi: Ylläpidettävyys pilvinatiiveissa arkkitehtuureissa

Project: Master's Thesis

Study line: Software and Telecommunication Technology

Page count: 61+1

Abstract: Goal of the thesis is to investigate how maintainability is addressed during the architectural design phase of cloud-native software development lifecycle. To this end, I conducted a survey among cloud architects of company Nordcloud, where I work. First I ascertain the perceived importance of maintainability. Then I categorize the suggested approaches for addressing maintainability and relate this to the respondents' years of experience in the target domain. Finally, I compare the survey results to approaches suggested in the literature.

Keywords: maintenance, maintainability, public cloud, cloud-native, cloud architect, architecture, software architecture, Master's Theses

Suomenkielinen tiivistelmä: Tutkielman tavoitteena on selvittää kuinka ylläpidettävyys huomioidaan pilvinatiivien sovellusten arkkitehtuurisuunnitteluvaiheessa. Tämän saavuttamiseksi suoritan kyselyn Nordcloud-yrityksen pilviarkkitehtien keskuudessa. Ensin määritän kuinka tärkeänä ylläpidettävyyttä pidetään. Sitten kategorisoin ehdotukset ylläpidettävyyden huomioimiseen, ja suhteutan ne vastaajien kokemuksen määrään kohdealueelta. Lopuksi vertaan kyselyn tuloksia kirjallisuudessa ehdotettuihin lähestymistapoihin.

Avainsanat: ylläpito, ylläpidettävyys julkinen pilvi, pilvinatiivi, pilviarkkitehti, arkkitehtuuri, ohjelmistoarkkitehtuuri, pro gradu -tutkielmat

Glossary

AD	Architectural Definition
BaaS	Backend-as-a-Service
CaC	Configuration as Code
CI/CD	Continuous Integration, Continuous Delivery/Deployment
CNA	Cloud-Native Application
FaaS	Functions-as-a-Service
IaC	Infrastructure as Code
MSA	Microservices Architecture
VM	Virtual Machine

List of Tables

Contents

1 Introduction

Goal of the thesis is to investigate how maintainability should be addressed during the architectural design of a cloud-native application. Maintainability is a quality attribute, which describes how easy it is to maintain and modify a given system. Software architecture, created by the software architect, formalizes the structural and behavioral foundation upon which the rest of the software will be built. Software development life cycle can be described with different life cycle models that prescribe a way to arrange and order the different activites of software development. Cloud-nativity means using the technologies and methods best suited for the cloud computing paradigm. I will expand upon the relevant terminology in chapter 2.

I aim to answer three research questions:

RQ1 How much importance do cloud architects place on maintainability during the architectural design phase?

RQ2 How to address maintainability concerns when architecting cloud-native applications?

RQ3 Do the recommendations from literature match the views of architects working in the field?

The main contribution of this thesis are the results of a survey I conducted in order to answer the two first research question. The survey was directed at cloud architects working at my company *Nordcloud*. When analysing results of the survey, I will first ascertain the perceived importance of maintainability. Then I will categorize the suggested approaches for addressing maintainability and contrast this to the respondents' years of experience in the target domain. Finally, I will compare the survey results to approaches suggested in literature.

I chose this topic for three reasons. First, the maintenance phase of software development life cycle is prevalent. It takes up the majority of total lifetime and costs. (Bass, Clements, and Kazman 1998). Second, choices made during the architectural design phase cascade into development and maintenance phases. Mistakes and oversights are slower and more expensive to correct later. (Bass, Clements, and Kazman 1998; Mumtaz, Singh, and Blincoe 2021). Lastly, the cloud-native approach is highly relevant in today's technology landscape. It helps reduce time-to-market and move costs from capital expenditure to operational ex-

penditure. Utilizing a public cloud platform also allows the company to leverage a highly scalable, reliable and secure infrastructure and a wide variety of easily integratable services. (Vettor, Pine, Jain, et al. 2022).

The thesis is structured as follows. In chapter 2 I dive into related research literature and other contemporary sources to define the necessary terminology. At the same time, I keep an eye out for best practices and other suggestions how to most effectively leverage the information in practice. Then in chapter 3 I describe how my own research is conducted in regards to the survey and analysis thereof. The results will be presented in chapter 4, and further discussed in chapter 5. I will close with conclusions in chapter 6.

2 Background

My main database for researching existing literature was the *JYKDOK* search engine for international articles ("JYKDOK international articles search" 2023). I decided to search keywords in abstracts, instead of full-text or only in title. In my opinion the full-text search was too lax, and title-only too strict. With these search terms I received between 4 to 473 results per search, which was sufficient:

- (Abstract:"cloud native" AND Abstract:maintainability)
- (Abstract:"cloud native" AND Abstract:architecture)
- (Abstract:"cloud native" AND Abstract:maintenance)
- (Abstract:architecture AND Abstract:maintainability AND Abstract:cloud)
- (Abstract:serverless AND Abstract:maintainability)
- (Abstract:serverless AND Abstract:operations)
- (Abstract:"cloud-native" AND Abstract:quality AND Abstract:attribute)
- (Abstract:software AND Abstract:"quality attribute" AND Abstract:maintainability)

I was unable to find full-text of some articles with promising abstracts. In case of one specific article I emailed the authors [1], and they kindly sent me a PDF of it. Additionally, I employed an ad-hoc snowballing from bibliographies of promising sources.

2.1 Architecture in software development

2.1.1 Software architecture

Every software system has an architecture. Bass, Clements, and Kazman (1998, 23) state that the architecture exists even if no-one present knows any details about it. They provide us a definition as follows:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

1. Bogner et al. (2018)

Next I will define the different parts of this definition, and adjacent terminology. A software system is a collection of software components organized to accomplish a specific function or set of functions (ISO/IEC/IEEE 42010:2007 2007, 3). These components — or elements — can be entities of varying size: objects, processes, libraries, databases, commercial software, and so on. In its most trivial case, we can consider the entire system to be a single component. This is indeed an architecture, but practically useless due to lack of necessary details (Bass, Clements, and Kazman 1998, 24). Koskimies and Mikkonen (2005, 53) define a component as an individual software unit that offers its services through well-defined interfaces. Implementation details each component are not considered part of architecture, but their interactions are. Without describing the interactions we end up with a ”bubbles and lines” diagram which is not sufficient to describe an architecture (Bass, Clements, and Kazman 1998, 24).

A software architect is a person, team or organization responsible for systems architecture (ISO/IEC/IEEE 42010:2007 2007, 3). A system can include more than one structure (Bass, Clements, and Kazman 1998, 23). Because ”structure” is an abstract concept, it can change depending on perspective. A architectural view models some aspect of the system (Koskimies and Mikkonen 2005). It is a representation of the whole system from the perspective of a related set of concerns, such as a specific stakeholder's specific concern (ISO/IEC/IEEE 42010:2007 2007, 3). A stakeholder is an individual, team or organization with interests in, or concerns relative to, a system. Examples of stakeholders include clients, users, the architect, developers, reviewers, and so on. (ISO/IEC/IEEE 42010:2007 2007, 3). Examples of architectural views include a scenario view, logical view, process view, development view, physical view, and modification view. They are partly overlapping: when a specific part of a system is changed, we often need to apply changes in multiple views. (Koskimies and Mikkonen 2005).

Architecture represents the first design decisions of a system (Bass, Clements, and Kazman 1998). It is difficult to get it right the first time, and can be laborious to alter down the road. In worst case, there is a need for an architectural change, which might result in changes across the entire system and changes to interaction between components. If we instead can limit the change to a group of components or only a single one, the change will be easier to carry out.

(Bass, Clements, and Kazman 1998, 31). Out of most design artifacts, architecture has most far-reaching consequences (Bass, Clements, and Kazman 1998, 31). It will place constraints on implementation and maintenance: will the system support the required functions; how easy is it to test, maintain, change and extend; how large data masses can it process, and so on (Koskimies and Mikkonen 2005).

The architecture might even have effect on the development organization by nudging towards a specific way to divide tasks to teams by structure of its components (Bass, Clements, and Kazman 1998, 31). This is somewhat contrary to Conway's law [2] which states that the technical structure of a system will reflect the social boundaries across the organizations that produced it. I suppose that in an agile organization the social structure is malleable enough to be impacted by a new system, when new teams need to be formed to facilitate the development.

According to Koskimies and Mikkonen (2005), architecture lies on a higher abstraction level than implementation. The lower abstraction level of a system manifests as source code (Bass, Clements, and Kazman 1998, 24). This might be named as "the implementation", or "design" as named in ISO/IEC/IEEE 12207:2017(E) (2017). Also ISO/IEC/IEEE 42010:2007 (2007, 4) states that architecture is conceptual instead of concrete. This goes hand in hand with the used vocabulary. With architecture, we are using the vocabulary of the solution, and are further away from the problem domain than during implementation (Koskimies and Mikkonen 2005).

When selecting a reference architecture or a fundamental architectural style, we are dealing with a generic solution that will be further specified later. An architectural style is a general, high-level guideline for structuring the components and their responsibilities (Bass, Clements, and Kazman 1998, 24). For example, client-server architecture is an architectural style. It is not an architecture in itself: terminology is generic or domain-specific, and the style is not specific enough to address the requirements set for your new system. Countless different kinds of architectures can be implemented with the same architectural style. (Bass, Clements, and Kazman 1998, 24).

2. https://en.wikipedia.org/wiki/Conway%27s_law

As we've read, architecture is of utmost importance for the implementation and future of the system. Because of this, it seems common sense to document it somehow. Bass, Clements, and Kazman (1998, 24) define an Architectural Description (**AD**) as a collection of products to document an architecture. They outscope the documentation of guiding processes from actual architecture, which means that an architecture can live independently of its description or specification. In contrast, ISO/IEC/IEEE 42010:2007 (2007, 3) mentions "... *and principles guiding its design and evolution*" when defining "architecture". The standard states that chosen architectural concepts, as well as considered alternatives, must be justified in the description. According to the standard, the description can be utilized at multiple different levels of abstraction, at each level emphasizing the details necessary for that level. If an AD is not available, it can be reverse-engineered from an existing system. This might be needed if, for example, it was originally developed without a description, or if the original architect is no longer available for consultation. (ISO/IEC/IEEE 42010:2007 2007, 7, 67).

A software architecture doesn't spawn out of nothing. The architecture is based on, among other things, the requirements placed on it (Bass, Clements, and Kazman 1998). Requirements codify the collective wishes and needs that various stakeholders want from the system. If requirements were the only input to software development, then an identical set of requirements would result in two pieces of identical architectures, even from two entirely unrelated organizations. (Bass, Clements, and Kazman 1998, 5–9).

As we can see, the organization, architect or architects, and the technical environment greatly influence how software systems are built (Bass, Clements, and Kazman 1998). The organization might have other existing products, architectures or assets that can be utilized. The organization has some individual structure, goals and long-term investments, and personnel with specific skills and availability. Especially the architects' background and experience is pivotal in deciding the course for the architecture. The architect might lean towards a familiar architectural style that they have had success with in the past, and vice versa with an unhappy experience. Their professional community can infuse into the new system a set of standard industry practices and software engineering techniques. (Bass, Clements, and Kazman 1998, 5–9).

The architecture cause feedback to its background factors, such as the organization, cus-

tomers, architects and the software engineering culture in general (Bass, Clements, and Kazman 1998). Because an architecture dictates the structure of a software system, it influences the structure of the development project with respect to team formation, schedules, and budgets. If the architecture is seen as successful, it might be used in future projects, and then the team structure might become a permanent part of the organization. The software might also offer a foothold in a certain market area, and this way the organization could update its goals to take advantage of this new possibility. Customers will likely grow fond of a good software product, which lowers the threshold for reusing a battle-tested architecture, because this is more economical than building a new one from scratch. The architects themselves gain more experience from this specific system, which means that the system's success or failure will affect architectural choices made in future projects. When looking through a wider lens, an especially novel and successful architecture might influence the software engineering culture as a whole, outside of the organization from which it originated from. Imagine, for example, the tides created by the first relational databases, spreadsheets, windowing systems, or WWW itself. (Bass, Clements, and Kazman 1998, 10–11).

2.1.2 Software development life cycle

In addition to every software system having an architecture, every software system also has a life cycle (ISO/IEC/IEEE 12207:2017(E) 2017, 17). A life cycle model is a framework that contains the processes, activities and tasks used during the life of the system, from initial requirements gathering to the termination of its use (ISO/IEC/IEEE 42010:2007 2007, 3). The specific life cycle stages depend on the chosen life cycle model, of which there are many to choose from: incremental, spiral, iterative, evolutionary, and so on. Usually there is an initial planning stage, where the need for a new software system arises. After the need is verified, requirements are gathered. Next there might be a prototyping phase where multiple architectures are rapidly mocked and evaluated. Once a rough architectural direction is chosen, an initial architectural definition is formed. After it is validated, there often follows an iterative cycle where analysis, design, implementation, verification, validation and delivery are repeated and intertwined. (ISO/IEC/IEEE 12207:2017(E) 2017, 18).

Koskimies and Mikkonen (2005) suggest an architecture-oriented software development pro-

cess. It highlights architectural design and evaluation as a crucial life cycle stage before moving onto detailed design and implementation. Architecture should be devised incrementally and iteratively from the requirements most relevant to architecture. Usually it is good to start from functional requirements, after which the non-functional qualities must be pursued. (Koskimies and Mikkonen 2005).

In this thesis I don't cling to any particular software lifecycle model. Instead I rely on the general notion that the initial architecture definition and architectural design happens early on in the life cycle of a software system. If using an iterative life cycle model, the requirements, architecture or detailed design might naturally need refinement also later on during operations and maintenance (ISO/IEC/IEEE 12207:2017(E) 2017). I will focus on the early choices and design decisions that influence the activities for the rest of the software lifecycle. We can call these architectural activities as "architecting", which comprises defining, documenting, maintaining, and improving an architecture, and certifying its proper implementation (ISO/IEC/IEEE 42010:2007 2007, 3). In addition to these architecting activities, Bass, Clements, and Kazman (1998, 12) mention initially creating a business case for the system, before any architecture is created.

The standard ISO/IEC/IEEE 12207:2017(E) (2017) defines processes related to architecture and design. Next I will make a distinction between two related but separate processes: the Architectural Definition process, and the Design Definition process. The purpose of the Architectural Definition process, as outlined in ISO/IEC/IEEE 12207:2017(E) (2017, 66), is threefold. First, the architect should create alternatives for a system's architecture. Then, we should choose one or multiple alternatives that properly frame the stakeholders' needs and fulfill the requirements placed on the system. Lastly, all of this should be described with a set of uniform architectural views.

At a lower abstraction level, the Design Definition process (ISO/IEC/IEEE 12207:2017(E) 2017, 71) is expected to provide enough detailed data and information about the system and its components for implementation to be in line with the architectural entities. It involves preparing for software system design definition by e.g. prioritizing design principles and characteristics, identifying and planning for necessary enabling systems and services, and obtaining access to those enabling systems and services. Then we should establish designs

related to each software element. This goes deep in the technical details, such as database structures, provisions for memory and storage, software processes, and external interfaces. (ISO/IEC/IEEE 12207:2017(E) 2017, 72). In this thesis I consider this realm of ”design” or ”detailed design” to belong to software developers instead of architects.

As we’ve discussed before, architecture is at a higher level of abstraction than design (ISO/IEC/IEEE 12207:2017(E) 2017). Architecture focuses on suitability, viability and desirability that the architecture is supposed to provide. Meanwhile, design is focused on compatibility with technologies and other design elements, and feasibility of implementation and integration. An efficient architecture is as design-agnostic as possible, to provide most flexibility in planning the said design. (ISO/IEC/IEEE 12207:2017(E) 2017, 71). Because ”design” is such an ambiguous word, in this thesis I will use it in accordance to the definition from ISO/IEC/IEEE 12207:2017(E) (2017) as an activity separate from architecting, unless used in conjunction with with ”architectual”, as in ”architectual design”.

As I mentioned above, according to ISO/IEC/IEEE 12207:2017(E) (2017, 71–72) selecting the technologies for implementation and considering compatibility with technologies also falls under the Design Definition process. In my experience with public cloud providers such as Amazon Web Services, Microsoft Azure, and Google cloud, implementation technologies are at least to some degree decided by architects instead of software developers. The architect wants to know which cloud platform is required or preferred by the customer, in order to reduce the technology space to a reasonable set of alternative services provided by the platform. The architect can then work with the available services to compose an architecture that best serves the stakeholders’ needs. The finer details, such as the choice of programming language or code-level design patterns, can be left to software developers, as also implied by the ISO/IEC standards.

Maintenance stage is a phase in software life cycle that is often considered less glamorous than working on greenfield projects and building the next software hit. Despite this, it is generally accepted that a substantial portion of the total cost of software is spent on maintenance. Bass, Clements, and Kazman (1998, 32) go as far as to say this figure is close to 80%. Obviously the actual numbers might be slightly different these days 25 years after their estimate, but the implication holds: considering maintenance efforts as early as possible in

the software life cycle can pay off. Mumtaz, Singh, and Blincoe (2021, 1) echo that architectural issues should be tackled in a timely manner, because the cost of fixing those kinds of problems later can be exceptionally high.

The standard ISO/IEC/IEEE 12207:2017(E) (2017) outlines the Maintenance process, which happens later in the software life cycle after Architectural Definition and Design Definition processes. Its purpose is to sustain the capability of the system to provide a service. This can be achieved by monitoring the system's capability to deliver services; recording incidents for analysis; taking corrective, adaptive, perfective and preventive actions; and confirming restored capability. In practice, preparing for maintenance should begin already during architecting. It is necessary to consider constraints placed on the architecture by maintenance-related requirements. This can be done, for example, by emphasizing encapsulation, modularity, and scalability. Documenting the architecture and system reduces the efforts required to reverse-engineer the system and components when a fix is needed. The decisions made during architecting can also affect the need and possibilities for e.g. remote diagnostics and maintenance, roll back, backup, and recovering data. (ISO/IEC/IEEE 12207:2017(E) 2017, 95–96).

Tasks performed during maintenance activities vary between systems and organizations. There is a need to review stakeholder requirements, complaints, events, incident and problem reports in order to identify the type of maintenance effort needed (ISO/IEC/IEEE 12207:2017(E) 2017). If the organization has adopted an iterative software life cycle model, software requirements might change during maintenance. This change in requirements can be a source for adaptive and perfective maintenance activities. In that case there will be a need to make corrections, improvements or other changes to software that is already deployed (ISO/IEC/IEEE 12207:2017(E) 2017, 97). As Bass, Clements, and Kazman (1998, 32) points out, having a good grasp of the architecture will help manage changes and reason about them.

2.1.3 General advice for achieving a good architecture

Because every system has an architecture, you can create an architecture through the simple method of trial and error. Unfortunately, an architecture built this way is unlikely to fulfill the requirements and goals placed on it (Bass, Clements, and Kazman 1998, 24). They outline some helpful advice for creating an architecture from the correct principles. These rules of thumb are divided into two categories: process-related and structure-related advice.

First we will delve into the process-related topics. As a general rule, there should be a single architect or a team of architects with a designated leader (Bass, Clements, and Kazman 1998, 15). Functional requirements and a prioritized list of quality attributes should be provided, so that the architecture can be steered towards achieving them. The architecture should be formally evaluated for quality attributes, and analyzed for applicable quantitative measures, such as maximum throughput. Having an architecture support iterative implementation makes integration and testing easier: a ”skeleton system” can prove that component communication works, even with otherwise minimal functionality. The architect is also responsible for defining, distributing, maintaining, and enforcing resource constraints to the responsible implementation teams. For example, there might be limits to network traffic or hardware performance that the teams must heed. A good documentation is also a must, with at least one static view and one dynamic view, written in a notation that all stakeholders can understand with minimal effort. Speaking of stakeholders, they should be actively participating in evaluating the architecture, to ensure their needs are being met. (Bass, Clements, and Kazman 1998, 15).

Next in line are the structure-related pieces of wisdom. Bass, Clements, and Kazman (1998, 16) state that modules should be well defined, with a clear separation of concerns and following the principles of information hiding. Their interfaces should also be well defined, and encapsulate any details that might change during implementation, such as choice of data structures. To improve modifiability, the modules that produce data should be kept separate from those that consume data. Between the modules, there should be only a small set of different ways of interaction, which is to say, the system should perform similar tasks in a similar fashion all around. This helps improve understandability, reliability, and modifiability, with the added benefit of shorter development time and conceptual integrity of the

architecture. Each quality attribute should be strived towards with well known architectural tactics that suit it. The architecture should not depend on any specific version of a commercial product or tool, or if it does, changing the product or tool should be both straightforward and cheap. If working close to hardware, changing the processor allocated for each task or process should be easy even at runtime. In case of parallel-processing systems, special care should be given to processes or tasks that touch multiple modules. (Bass, Clements, and Kazman 1998, 16).

2.2 Maintainability

2.2.1 Quality Attributes

We of course expect that the software system fulfills all its requirements, gives us correct and accurate results, and that it interacts with other systems in the expected manner (Bass, Clements, and Kazman 1998, 76). Additionally, there usually is an implicit or even explicit goal of creating software of high quality. According to Gorla and Lin (2010, 602), this quality can be influenced by factors that are organizational, technological, or end user-related. Organizational factors can include the IT budget, number of people in system development in that organization, and quality of documentation. Level of user involvement and user training the systems are end user-related factors. In contrast, technological factors include experience and skill level of the development staff, and type and suitability of development method, programming language or database model (Gorla and Lin 2010, 602). Few factors outside the technological category are relevant in this thesis, which is why I mostly – although not exclusively – focus on factors that can be influenced by the software architect.

Quality attributes are used to describe specific aspects of quality as it relates to software. From management perspective, we are likely interested in business-related attributes and metrics such as cost and effort of development work, productivity estimation, and time-to-market (Arvanitou et al. 2017, 60; Bass, Clements, and Kazman 1998, 76). In this thesis I only deal with these topics at a superficial level, as they cannot be directly quantified based on architecture alone. Bass, Clements, and Kazman (1998, 76) suggest that there are two categories of quality attributes that instead can be measured on architectural level: attributes

that can be observed by running the system, and those that cannot. The first category tells us how well the system fills its behavioral requirements during runtime. Quality attributes that indicate this include (Bass, Clements, and Kazman 1998, 79–81; Li et al. 2021, 13; Arvanitou et al. 2017, 60):

- performance
- security
- availability
- functionality
- usability
- monitorability
- reliability
- safety
- fault prediction
- defect proneness

The second category of quality attributes — those that cannot be observed by running the system — tell us how easy the system is to test, integrate and modify (Bass, Clements, and Kazman 1998, 81). It contains, among others, these attributes:

- modifiability
- portability
- reusability
- integrability
- testability

Other authors use related terms ”functional” and ”non-functional requirements”. Bass, Clements, and Kazman (1998, 76) consider this dichotomy useless. They argue it suggests that all attributes can be divided strictly into those that model the system’s ability to perform the actions it is expected to, and ”everything else”. The latter non-functional category acts as a bucket. This too easily leads to a situation where some qualities are given less consideration, leading to a diminished quality of the system as a whole. In fact, many quality attributes are intertwined to behavior of the system. For example, it might be impossible to reach lightning

fast performance with a feature that processes very large images. On a higher level, though, Bass and others consider functionality to be orthogonal to quality attributes. If this wasn't the case, implementing a certain functionality would always dictate the system's security, usability, or other attributes. The program might function perfectly, but it might also be difficult to modify or costly to build. (Bass, Clements, and Kazman 1998, 77).

Instead, architectural choices made by the architect decide the level of quality possible for the system (Bass, Clements, and Kazman 2003, 72). Quality must be considered at every stage of design, implementation and deployment. Different quality attributes represent themself differently at different stages. For example, some usability decisions, such as details of a UI, are not architectural at all. Modifiability, on the other hand, is primarily architectural. Modifiability relates to how a system is split into components, and a system is modifiable if changes do not require touching multiple components. Performance depends on both architectural and non-architectural considerations. Architect can influence how functionality is divided between each component, and how they communicate. But the finer details are left to developers, who are usually responsible or detailed design. An example of this is the choice of an algorithm and turning it into source code. Another thing to consider is that quality attributes do not exists in a vacuum, but instead influence each other in good or bad (Bass, Clements, and Kazman 1998, 78). For example, improved monitorability likely enables improved testability, which in turn leads to an increase in security (Li et al. 2021, 18). The architect must prioritize the quality attributes and decide which ones to emphasize (Bass, Clements, and Kazman 1998, 129).

2.2.2 What is maintainability, and why is it important

The quality attribute maintainability describes the efforts needed to fix errors or to make enhancements to the system (Gorla and Lin 2010, 603). Based on their mapping study on design-time quality attributes and metrics, Arvanitou et al. (2017) found that maintainability is the most often researched quality attribute in most stages of the software development life cycle. It receives interest in design, architecture, implementation and maintenance stages. Maintainability is especially relevant also in other stages such as project management, requirements, or testing stages. (Arvanitou et al. 2017, 61–62).

Like many other quality attributes, maintainability can be further subdivided into more specific quality attributes, some of which we have discussed before. ISO/IEC 5055:2021(E) (2021, 233) lists modularity, reusability, analyzability, changeability, modification stability, testability, and compliance as components of maintainability. Many of these are also echoed by Arvanitou et al. (2017, 61). Some other categories, such as understandability, are sometimes related to maintainability. Arvanitou et al. (2017) consider it a separate quality attribute for two reasons. First, some other quality models make a distinction between maintainability and understandability. Second, a unique research community has already gathered around understandability as a standalone concept. (Arvanitou et al. 2017, 61).

There is no single metric ”maintainability of the software system”, because different components of the system likely have different purposes and implementations (Broy, Deissenboeck, and Pizka 2006, 26; Bass, Clements, and Kazman 1998, 192). As such, a phrase like ”the system is highly maintainable” doesn't tell anything practical about the system, because quality attributes receive their meaning from surrounding context. For example, the software might be highly modifiable in relation to specific classes of changes. (Bass, Clements, and Kazman 1998, 192).

Focusing on maintainability can pay back the cost invested into it. It influences how much money and time needs to be spent on maintenance tasks (Broy, Deissenboeck, and Pizka 2006, 21). If it is not considered, cost of change can be unnecessarily inflated (Vale et al. 2022, 2). Personnel requirements have to also be considered, because someone needs to perform the necessary updates and fixes. Wiggins (2011) warns that if maintenance work is left to the one single team of software developers, the time is often deducted from development of new features.

If a system is not maintainable, it will be more difficult to address other problems, such as those related to performance or reliability (Bouwers and Deursen 2010, 46). If different services are highly coupled, one of them cannot be altered without changing others that are depending on it (Vale et al. 2022, 2). At the business level, maintainability of an important system defines the organization's ability to adapt their business processes to changing market circumstances, and to implement innovative products and services (Broy, Deissenboeck, and Pizka 2006, 21).

Even if the software itself stays unchanged for years, it can break if and when its environment changes. This phenomenon is called software rot [3]. In other words, software rot happens when a system's assumptions go out of date. A good example of this is the infamous "Y2K" event which resulted in uncountable bugs when two-digit year counters underwent wrap-around at the start of the new millennium. Even outside such one-off situations, software can touch the underlying operating system and network infrastructure in many places: library versions, directory paths, IP addresses and hostnames, and so on (Wiggins 2011). The Jargon File [4] entry about software rot advises to employ robust design to deflect such problems. Based on what we've discussed in this chapter, I consider taking care of maintainability part of robust design.

2.2.3 How to measure maintainability

As discussed before, problems discovered early in the software development life cycle are easier to address (Bass, Clements, and Kazman 1998). At that point you still have a chance to adjust requirements, specification or design with relatively low cost. Software quality cannot be "appended" to the project afterwards, but must instead be baked in deliberately. Qualities of a system can be anticipated by evaluating the architecture. This means we should evaluate the architecture as soon as feasible. To evaluate a certain quality attribute, we must make sure we have the necessary architectural descriptions at hand. For example when evaluating modifiability — a subcategory of maintainability — we need at least a decomposition that shows us the different modules and how tasks are distributed between them. (Bass, Clements, and Kazman 1998, 32, 190–191).

It is necessary to decouple quality attributes from effort estimations in our daily work. Moses (2009) points out that maintainability of a software module is intended to signify possible challenges posed by code for maintenance tasks. Maintainability cannot give you an estimate of how difficult a certain maintenance change will be or how long it will take. Unless the same change has been performed in exactly the same circumstances before, duration and

3. http://www.catb.org/jargon/html/S/software-rot.html.

4. The Jargon File is a "comprehensive compendium of hacker slang illuminating many aspects of hackish tradition, folklore, and humor." http://www.catb.org/jargon/html/online-preface.html

possibility of success of a code change are only subjective estimates. Moses suggests two methods for estimating practical maintenance tasks. It can be given on an ordinal scale of e.g. 1-5 instead of an absolute time estimate. If a concrete estimate is necessary, two experts should voice their opinions, from which we can calculate an expected duration with confidence limits. (Moses 2009, 204).

Vice versa, a prediction system for maintenance effort is not a prediction system for the quality attribute maintainability (Moses 2009, 206). Each maintenance effort includes factors that influence the duration of the activity. These factors include availability of documentation, amount of testing after the change, and skills of the maintenance team. Because it is obviously impossible to carry out and measure all possible maintenance activities for a system, the effort can only be measured from a subset of all possible maintenance activities. This means that, even in the best scenario, a system's maintainability can be estimated only for activities that are similar to ones that have already been performed before to this system. In the worst case, the measured effort can be misleading, if measured activities do not represent actions that are carried out in practice. (Moses 2009, 206).

Broy, Deissenboeck, and Pizka (2006) state that an evaluation criteria must be well-founded and checkable. Evaluating a whole software system is quite involved, and requires multiple kinds of evaluation methods. Source code can be automatically checked for syntactical properties. Other methods are semi-automatic, requiring some human actions. The rest of the methods are thoroughly manual, such as reviewing documentation, or evaluating whether the system employs correct data structures (Broy, Deissenboeck, and Pizka 2006, 22). Our focal point of interest, maintainability, is a high-level quality attribute, because it cannot be directly calculated from metrics (Arvanitou et al. 2017, 60).

However, there are ways to measure maintainability indirectly. In their study about designing microservice systems using patterns, Vale et al. (2022, 9) found that the participating industry experts do not measure quality attributes directly. Instead, the participants measured quality attributes as a sum of lower-level attributes or processes related to the quality attributes in question. These lower-level attributes include measuring code coverage, time taken to resolve issues, and analysis of architectural debt, which all are indirect metrics for maintainability. Vale and others reason that this might be because lower level attributes are

easier to measure and check directly than maintainability, which is in line with Arvanitou et al. (2017). Vale et al. (2022, 9) also postulate that subjective understanding of quality attributes is very much personal and often vague. This makes it difficult to mutually agree on a definition inside your organization. Instead, participants rather liked to think of techniques and tools they used to tackle the related problems. (Vale et al. 2022, 7–10).

Automatic methods for assessing maintainability of a system usually limit themselves to the source code level. Such methods cannot be used to to evaluate an architecture before implementation has taken place. They are however useful after the fact, for example when following an iterative life cycle model. The standard ISO/IEC 5055:2021(E) (2021, 1–2) points out that faults in architectural or design practice can also be detected by statically analysing design specifications. This requires that the specification is written in a design language with a formal syntax and semantics. The standard presents a long list of "Automated Source Code Quality Measures" (ASCQM) that provide strong indicators for the quality of a software system in each area. In my opinion, the following maintainability weaknesses are related to architecture (ISO/IEC 5055:2021(E) 2021, 38–47):

- invocation of a control element at an unnecessarily deep horizontal layer (layer-skipping call)
- invokable control element with excessive file or data access operations
- invokable control element with large number of outward calls (excessive coupling or fan-out)
- modules with circular dependencies

The standard also provides multiple detection patterns (ISO/IEC 5055:2021(E) 2021, 2). They help detect structural weaknesses from code. One pattern can be used to detect multiple weaknesses. I would list the following maintainability detection patterns as related to architecture (ISO/IEC 5055:2021(E) 2021, 48–54):

- limit number of data access
- limit number of outward calls
- ban circular dependencies between modules

Arvanitou et al. (2017, 63) note that there are two categories of metrics for quantifying qual-

ity attributes: those that can be evaluated through a single metric, and those evaluated with multiple. Arvanitou et al. (2017, 66) list the following metrics relating to the architectural stage:

- cyclomatic complexity
- coupling factor
- coupling between objects
- number of modules
- information flow (fan-out)
- non-functional coverage
- absolute adaptability of a service
- relative adaptability of a service
- mean of absolute adaptability of a service
- mean of relative adaptability of a service
- level of system adaptability

Most of the tools to calculate these metrics are unnamed, likely created in academia by research teams, and not available for public use. Maintainability is often connected to metrics, but is not always quantified as their function. Evaluating a quality attribute with a single function is non-trivial, which is why few research have suggested such a function for high-level attributes like maintainability. (Arvanitou et al. 2017, 67).

In regards to software, a ”smell” is considered synonymous to antipattern, flaw, or anomaly (Mumtaz, Singh, and Blincoe 2021, 1). In their paper about architectural smells detection, Mumtaz, Singh, and Blincoe (2021, 20) found that research tends to focus on dependency smells, which are usually connected to coupling issues in the architecture. Most detection tools can detect common dependency smells. The researchers describe the different approaches to detect architectural smells in a software system (Mumtaz, Singh, and Blincoe 2021, 8–14):

- graph-based
- design structure matrix
- model-driven

- code smells analysis
- reverse engineering and history-based
- search-based
- visualization
- rules-based

Graph-based approach depicts system's components as nodes, and their relationships as edges (Mumtaz, Singh, and Blincoe 2021). This makes it easy to notice problematic relationships between them. A design structure matrix is a two-dimensional matrix to represent the structural relationships of the software. This can be used to represent complex architectural components and their relationships. Model-driven approach represents structure and behavior of a system using abstractions and modeling. Search-based methods such as genetic programming are suitable for especially large datasets, and a visualization approach can help us understand large software systems with multivariate and multidimensional data. (Mumtaz, Singh, and Blincoe 2021, 9–14).

Rules-based approach takes advantage of metrics, their thresholds, and predefined frameworks, heuristics or guidelines (Mumtaz, Singh, and Blincoe 2021, 8). It is related to code smells analysis, which combs through source code to identify code smells in order to detect architectural smells from them. This approach can find a correlation between subjective experiences and economical facts of the system (Broy, Deissenboeck, and Pizka 2006, 22). Downsides to a rules-based approach is that it concentrates on features that can be calculated automatically from source code, which restricts it to syntactical factors. Automatic checking is difficult to employ when scrutinizing suitability of data structures, or quality of documentation. Standards and guidelines might be too generic to be evaluated. This is a problem if there is no way to concretely check the impact or value of an attribute, such as modifiability. Some guidelines might also lack reasoning for the presented advice: consider the proposition to keep method length to under 30 lines of source code. Why 30 lines exactly, and not 25 or 37 instead? Without a solid argument, the advice will likely stay unheeded. The approach also requires the user to additionally consider the organizational context for the automatically calculated result, or else it is useless in evaluating maintainability of the system as a whole. (Broy, Deissenboeck, and Pizka 2006, 22).

2.2.4 How to improve maintainability

There will be less need to reverse-engineer systems and elements during maintenance work if we write down documenting the system as an explicit requirement (ISO/IEC/IEEE 12207:2017(E) 2017, 96). Utilizing the available standards and system development methodologies helps create systems that are easily understandable. This results in systems that require less effort to maintain. (Gorla and Lin 2010, 608).

Broy, Deissenboeck, and Pizka (2006) remind us to separate ”facts” from ”acts”. An activity, or act, is something that we try to do. In context of software maintenance, example acts can include concept location, impact analysis, coding, and modification. A fact is some circumstance surrounding the system, such as readability of documentation, processes used in the organization, or availability of debugging tools. Maintainability also depends on things outside of the product itself. For example organizational facts are not part of the product, and often overlooked when assessing maintainability of a system. Skill of the maintainers plays a key role in maintainability. A high turnover rate can make a system more difficult to maintain, if the company loses a large share of its developers to a competitor. Well-defined processes, such as a configuration management process, can help create a quality product. Trustworthy tools like debuggers, and visualization and refactoring tools are critical for performing effective maintenance. (Broy, Deissenboeck, and Pizka 2006, 25–26).

Bass, Clements, and Kazman (1998) collate that the most important thing in improving non-runtime attributes like modifiability and testability is to enable and facilitate making changes to the system. The system likely suffers from poor modifiability if it relies on a shared memory solution with global variables accessible to all components, where a change to data format could require changes to all components (Bass, Clements, and Kazman 1998, 89). A change should only affect a small number of components. ISO/IEC/IEEE 12207:2017(E) (2017, 96) concurs that systems which highlight encapsulation, modularity, and scalability can be simpler to maintain. Achieving this requires foresight into what kinds of modifications will be necessary in the future. The foresight can be achieved by an experienced architect with available change histories for this and related systems. (Bass, Clements, and Kazman 1998, 118).

To improve maintainability, Vale et al. (2022, 7) suggest tracking bug reports with *Slack* and *Jira*. Static code analysis tools such as *SonarQube* and *Codacy* help on the source code level. Code reviews, automated pipelines, and domain-driven design are also beneficial. According to their research, there are design patterns one can use to gain an edge with maintainability. Strangler, Ambassador, External Configuration Store, Gateway Offloading, Backends for Frontends, Pipes and Filters, and Static Content Hosting patterns were reported by at least one interviewee to have improved maintainability of a system they have worked with. (Vale et al. 2022, 4–6). Design patterns create a common language between designers, to share an understanding of solutions that have been found to work well.

Related to design patterns, Bass, Clements, and Kazman (1998) talk about ”Unit Operations”, which are general design operations or techniques that allow the architect to achieve quality attributes. Unit operations have their roots in the engineering world, and are at a higher level of abstraction than design patterns. Examples of unit operations are Separation, Uniform Decomposition, Replication, Abstraction, Compression, and Resources Sharing. The unit operations Separation and Resource Sharing can have a positive effect on maintainability by improving modifiability. In Separation, functionality is detached as its own component with a well-defined interface to its surroundings. This has numerous advantages: it enables distribution, which in turn enables parallelism; it’s easier to divide development work to development teams; and it improves modifiability and portability. It springs to my mind that the Separation unit operation could also be used to create layers in a layered architecture. Resource Sharing means that data or services are encapsulated and then shared between multiple independent consumers. Examples of this are databases, blackboards, and servers in a client-server-architecture. This unit operation decreases coupling between components, which leads to improved integrability, portability and modifiability. (Bass, Clements, and Kazman 1998, 123–126).

The *Heroku* cloud application platform (Wiggins 2011) utilized ”Explicit Contracts” to maintain a strong separation between the platform and the applications running on it. This results in what they call ”erosion-resistance”, which is synologous to improved maintainability. An example of an explicit contract is that in a *Ruby* application, the developer must use a *Gem Bundler* and a *Gemfile* to declare dependencies of ther application. In a *Node.js* applica-

tion this is achieved via *package.json* file instead. Other contracts dictate how environment variables should be used, how to implement logging, and how to tell the platform how you want your application to be launched. Adhering to these explicit contracts allows Heroku to change the infrastructure and platform without breaking applications running on it. (Wiggins 2011).

Different quality control activities should be performed at different intervals (Broy, Deissenboeck, and Pizka 2006, 25). To avoid redundancy, copy-pasted sections of new code should be checked up to daily. Then again, it is enough to review documentation at certain points of the development process, and not even weekly. This is mainly because manual quality checks are laborious, and thus costly. Whenever feasible, manual review should be supported or replaced with automated tools to as high a degree as possible. This makes good quality check tooling valuable, because we can receive high-quality assessments more often. (Broy, Deissenboeck, and Pizka 2006, 25).

A system might be labeled as a ”legacy system” due to factors that do not negatively affect usage or functioning of the software (Broy, Deissenboeck, and Pizka 2006, 22). Such factors can be the coding style, or the implementation language. A label of ”legacy” hints at unmaintainability, and the will to replace the system with something else. Broy, Deissenboeck, and Pizka (2006, 22) argue that a legacy system should not be replaced unless the new system increases business value or maintainability. Li et al. (2021, 18) concurs: a new architecture could mitigate some problems of the legacy system, but implementing it properly will cost time and effort.

2.3 Cloud-nativity

2.3.1 What is cloud-nativity

The cloud is a distributed architecture of individual cloud-native services, providing resources as services in a tiered fashion to construct a full technology stack from hardware to middleware platforms to applications (Pahl, Jamshidi, and Zimmermann 2018, 1). Birth of the public cloud can be dated back to 2006, when *Amazon Web Services* launched their *Simple Storage Service* (S3) and *Elastic Compute Cloud* (EC2) (Kratzke and Quint 2017,

1). The so-called first wave of cloud computation meant that companies replaced their own data centers with virtual machines (**VM**s) running on data centers owned by cloud platform providers (Gannon, Barga, and Sundaresan 2017, 17). While revolutionary, it was still difficult to support both scalability and security at the same time, and there was no cloud-based tooling for data, event or error correction to be used in debugging. (Gannon, Barga, and Sundaresan 2017, 18).

”Cloud-native” is a term often used, but rarely elaborated beyond ”we used the cloud instead of our own data center” (Gannon, Barga, and Sundaresan 2017, 17). Terminology and related technology have gone through many steps to get where we are now, touching Service Oriented Architecture (SOA), VMs, cloud computing, containers, and microservices (Kratzke and Quint 2017, 3). Cloud-nativity is an approach that only makes sense on top of cloud infrastructure, which means that the physical environments have been virtualized, and infrastructure is disposable. When using disposable infrastructure, units of infrastructure are created, scaled, and destroyed quickly with help of automation. There is a fitting metaphor of ”cattle vs. pets”, where disposable infrastructure is the cattle: all instances are identical, and if one of them requires an update or repairs, it will be replaced instead of fixed. (Vettor, Pine, Jain, et al. 2022).

Kratzke and Quint (2017, 8) searched ”cloud-native” from *Google Trends* for an overview of its usage. The term was widely used around 2007, around the first wave of cloud computation. Usage tapered after that, but around 2015 the term enjoyed newfound interest. The researchers admit that Google Trends is not the most reliable of metrics due to its sensitivity to industry buzzwords. These results, however, correspond to the results of their research. ”Cloud-native” was first mentioned in academic literature in 2012 [5] [6]. (Kratzke and Quint 2017, 8).

Vettor, Pine, Jain, et al. (2022) offer a definition for ”cloud-native”:

> Cloud-native architecture and technologies are an approach to designing, constructing, and operating workloads that are built in the cloud and take full ad-

5. Andrikopoulos, Fehling & Leymann, 2012. Designing for CAP - The Effect of Design Decisions on the CAP Properties of Cloud-native Applications

6. Garca-Gmez, et al., 2012. 4CaaSt: Comprehensive Management of Cloud Services through a PaaS

vantage of the cloud computing model.

In my opinion this definition is vague, with the main conceptual burden carried by ”cloud” and ”cloud computing model”. Patrizio (2018) offers additional details to the definition: cloud-native is a modern way to build and run software systems, and leverages the elasticity, scalability and resilience offered by cloud computation. A still more specific definition comes from Kratzke and Quint (2017), whose systematic mapping study aimed to collate the research trends around cloud-native software development practices. They also wanted to define the Cloud-Native Application (**CNA**), for which the resulting definition is as follows (Kratzke and Quint 2017, 13):

> A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.

From this we can conclude that a CNA is not merely a distributed system, but especially distributed, horizontally scaling, resilient, and load-adaptive (Kratzke and Quint 2017, 13). Scalability is at the heart of cloud-native software (Noval et al. 2022). A CNA must be more more than ”available via the internet”: it must achieve global scale by serving thousands of concurrent users (Gannon, Barga, and Sundaresan 2017, 17). If you think of global, culture-penetrating companies, you might come up with *Netflix*, *Spotify*, *Uber*, *Airbnb*, *Facebook*, or *Twitter* (Gannon, Barga, and Sundaresan 2017; Patrizio 2018).

Companies such as *Microsoft*, *Google*, *Amazon*, *Oracle*, *IBM*, *Alibaba*, *Heroku*, and *Redhat* provide public cloud platforms for other companies to build cloud software on. They are called Cloud Service Providers (**CSP**). CSPs' own software products might utilize the CSP's own infrastructure, which makes sense for simplicity of logistics. Some services such as *AWS Kinesis*, *DynamoDB*, *SQS*, and *Amazon Redshift*; *Microsoft Azure CosmosDB* and *Data Lake*; and *Google BigQuery* are themselves cloud-native under the hood, and used to build CNAs. (Gannon, Barga, and Sundaresan 2017, 17).

CNAs share some common features. They are expected to be always running. At the same

time, a CNA must expect the physical infrastructure to vary, which leads to related intermittent errors (Gannon, Barga, and Sundaresan 2017, 17). This is different from single-machine applications, where we can usually assume that the platform — which consists of hardware and an operating system — stays the same during application runtime and always works. For CNAs, the reality is that once the application starts receiving a sufficient number of requests, something is always either breaking or broken. It is necessary to architect the software in such a way that updates and testing are possible without disruptions to the production instance of the software. Security should also be an integral part of the architecture, due to multiple small components, and a need to manage access on multiple levels of the software. A simple firewall is not enough. (Gannon, Barga, and Sundaresan 2017, 17).

CNAs utilize technology and techniques suitable for dynamic cloud environments. These cloud environments can be either public, private or hybrid (Noval et al. 2022). The techniques include microservices, containers, service meshes, immutable infrastructure, declarative APIs, continuous delivery technology, backing services and automation (Noval et al. 2022; Patrizio 2018; Vettor, Pine, Jain, et al. 2022). Methodologies such as DevOps and agile are often adopted when building CNAs (Patrizio 2018). Other CNA development methodologies are often based on design patterns (Kratzke and Quint 2017).

The three most common ways to build CNAs are microservices architecture, fully managed high-abstraction services, and serverless computing (Gannon, Barga, and Sundaresan 2017, 17). In fully managed services, the service handles everything outside of business logic: infrastructure, management, scaling, and so on. Microservices architecture is the most common approach. Serverless computing means utilizing CSP-provided services to build software without VMs and sometimes even without containers. (Gannon, Barga, and Sundaresan 2017, 17). I will discuss microservices and serverless in the following chapters.

2.3.2 Microservices

Microservices architecture (**MSA**) is an increasingly accepted and adopted architectural style, and is much used in modern softwares (Vale et al. 2022, 10). It was the first major style for cloud-native applications, dating back to 2013 (Gannon, Barga, and Sundaresan

2017, 18). Scalability is important when moving to a cloud-first paradigm, and MSA can help overcame the limitations of traditional monolithic systems related to scalability (Vale et al. 2022, 10). Monolithic software is often built using layered architecture, relational databases shared across all services, and is run as a single process (Vettor, Pine, Jain, et al. 2022). MSA on the other hand encourages you to implement small-scale, independently distributed services instead of tying up all functionality into a single monolith (Li et al. 2021, 1). When using MSA, you will disassemble the application into its basic building blocks by encasing a single functionality into a single service (Gannon, Barga, and Sundaresan 2017, 18). This also allows for technological heterogeneity, when different services can use the technologies that best allow it to reach its goal (Li et al. 2021, 18). Together these services form the complete application (Vettor, Pine, Jain, et al. 2022).

A single microservice implements a specific business functionality as part of a larger domain context (Vettor, Pine, Jain, et al. 2022). This context should be limited, giving the service limited responsibility, with only a small number of dependencies on other services (Gannon, Barga, and Sundaresan 2017, 18). According to Vettor, Pine, Jain, et al. (2022), the service should contain the necessary logic, state, data, external dependencies and programming platform (Vettor, Pine, Jain, et al. 2022). Meanwhile, Gannon, Barga, and Sundaresan (2017, 18) recommend that each service should pursue statelessness, where the state is not saved into the microservice itself, but in another service, such as a database, cache or directory. In my opinion these differing views on handling state are not conflicting. Absolutes are harmful, and as prescribed in the CNA definition by Kratzke and Quint (2017, 13), the main idea is to isolate state in a *minimum* of stateful components.

The microservice should run its own process independently of other active services, and communicate with them using standard messaging protocols, such as *HTTP/HTTPS*, *gRPC*, *WebSockets* or *AMQP*. Every microservice should be able to be separately managed, replicated, scaled, updated and deployed (Gannon, Barga, and Sundaresan 2017, 18). Vettor, Pine, Jain, et al. (2022) elaborate that independent scaling enables a more accurate control over the system. At the same time this lowers the total costs involved, because there is no need to scale services with lower usage alongside those that need it. Autonomous development and deployment of the separate services removes the need to wait for e.g. quarterly

release schedule to roll out an anticipated feature or fix. Independent deployment also reduces the risk of total system malfunction, as the changes are isolated to smaller context. (Vettor, Pine, Jain, et al. 2022). Security must also be considered. One microservice should only have access to select few other microservices, and the target service must verify access rights of the requester service. Using role-based access control such as *OAuth*, *AWS IAM*, or *Azure RBAC* can be instrumental in achieving this. (Gannon, Barga, and Sundaresan 2017, 18).

Gannon, Barga, and Sundaresan (2017, 18) remember a time when a need arose to pack microservices into spaces smaller than an entire VM image. When using containers, the code, dependencies, and runtime are packed into a container image (Vettor, Pine, Jain, et al. 2022). The images are then stored in a container registry, where they can be fetched when deploying a service that uses them. It is possible to run multiple containers in the cloud on a single VM, and achieve startup times even lower than one second (Gannon, Barga, and Sundaresan 2017, 18). Other reasons to select containers as a basis for your application include portability, and lack of a need to configure every environment separately with the required frameworks, libraries, and runtime engines. This helps to achieve uniform environments with greater accuracy and speed. (Vettor, Pine, Jain, et al. 2022). Companies such as *Docker* popularized the container technology for use in MSAs (Gannon, Barga, and Sundaresan 2017, 18). According to Vettor, Pine, Jain, et al. (2022), Docker is the most popular container vendor, and it has become industry standard for packaging, deploying and running cloud-native software.

Containers are often arranged in a service mesh, using a container orchestration solution like *Kubernetes*, *Apache Mesos*, *Docker Swarm*, *Azure Service Fabric* or *IBM Blue Container Service* (Gannon, Barga, and Sundaresan 2017, 18). The orchestrator takes care of things like scheduling, affinity, health monitoring, failover, scaling, networking, service discovery, and rolling upgrades (Vettor, Pine, Jain, et al. 2022). Scheduling means automatic provisioning of the container instances. Affinity and anti-affinity describe how far away the containers are from each other, measured in respect to hardware topology. Health monitoring allows for automatic detection and recovery from failures. If a failure happens, a failover process will automatically provision the failed instance to a healthy host. As we can extrapolate from

previous definitions of the word, scaling means automatically adding or removing container instances depending on the demand. The orchestrator handles the networking overlay for container communication. Service discovery makes it possible for containers to find each other. Lastly, the orchestrator coordinates incremental updates and rollback of problematic changes to achieve rolling upgrades. (Vettor, Pine, Jain, et al. 2022).

Jumping aboard the MSA bandwagon is not all sunshine and roses. Li et al. (2021, 18) point out that if the monolith is complex to begin with, componentizing it will bring this inner chaos on display. The end result might still be complex, even if the complexity is now slightly different from previous state of affairs. Gannon, Barga, and Sundaresan (2017, 20) remind that we need to allocate efforts to manage computation resource clusters in order to run our services. As I see it, while the container orchestrator service take care of automatic operations day-to-day, the initial setup is non-trivial, and monitoring is still required.

Vettor, Pine, Jain, et al. (2022) presents some challenges for MSA systems. I'd like to reframe these challenges as things to consider when architecting such systems, rather than strictly negative aspects of the MSA approach. The listed challenges are most often related to communication, resiliency, or distributed data. Firstly, because microservices need to communicate via APIs and network protocols instead of in-process, the architect should account for network congestion, latency and intermittent errors (Vettor, Pine, Coulter, Schonning, et al. 2022). It is especially important to retry failed requests, which is made easier if all state-altering requests are idempotent. There will also be some overhead for every service from serializing, deserializing, decrypting, and encrypting messages. (Vettor, Pine, Coulter, Schonning, et al. 2022).

While the cloud platform will automatically handle most of the resiliency issues that CNAs face, the architect still needs to consider them while designing the system (Vettor, Pine, Coulter, Wenzel, et al. 2022). In my view, this means minimizing disruptions caused by A) the issues themselves, and B) the actions that the platform takes to respond to the issues. The platform might restart and migrate a failed process to a new host. A container orchestrator might be in the middle of a rolling upgrade, or move the service from one node to the next, for one reason or another. (Vettor, Pine, Coulter, Wenzel, et al. 2022).

When following the MSA model to the letter, we will end up with distributed data. Because all data is no more in a single shared database, it will be more difficult to compose queries across multiple services (Vettor, Pine, Coulter, Victor, et al. 2022). The architect must also remember that the ACID principles do not hold across the entire system at any one point in time. This is because propagating a change of state through multiple data stores will take some time. On the plus side, each data store can be individually scaled and developed without direct effect on other services. Each of them can select a data model that best suits that specific service: relational, document, key-value, graph, or some other database paradigm. This boosts the agility, performance, and scalability of the system. (Vettor, Pine, Coulter, Victor, et al. 2022).

2.3.3 Serverless

Cloud-nativity doesn't constrain itself to MSA. Another relevant approach is striving for the so-called serverless architecture. The title "serverless" is something of an oxymoron, because there is a server, but it's not you who is responsible for it. Though it might mean zero system administration work for you, the abstraction might leak at times, and then you'll become aware that a Systems Administrator is supporting your application somewhere. Broadly speaking, this can be achieved in two ways: Backend-as-a-Service (**BaaS**) or Functions-as-a-Service (**FaaS**). (Roberts 2018).

With BaaS, you manage server-side logic and state by significantly or fully incorporating third-party cloud-hosted applications and services (Roberts 2018). These services handle everything else but your business logic: infrastructure, management, scaling, and so on. When such services are combined, we will end up with an application that fulfills the criteria for cloud-nativity (Gannon, Barga, and Sundaresan 2017, 20). Sometimes BaaS is used with the prefix "Mobile" BaaS, due to their popularity with application backends written for mobile devices. Examples of BaaS are *Parse* and *Firebase* databases, and *Auth0* and *AWS Cognito* authentication services. (Roberts 2018).

When going down the FaaS route instead, you use stateless, event-triggered, ephemeral, fully managed compute containers to run your server-side logic (Roberts 2018). Vendor

handles all the underlying resource provisioning and allocation by creating and destroying FaaS instances based on runtime need. FaaS instances are typically triggered as a response to inbound HTTP requests. Because you don't have control of the lifecycle of individual instances, you must store any persistent data to an external service, such as a database. While the management of the underlying compute resource is abstracted away to a very high degree, there's still monitoring, deployment, networking, support, and production debugging to handle (Roberts 2018). Examples of FaaS services are *AWS Lambda, Azure Functions, Google Cloud Functions* and many others. They are often paired with serverless services such as *AWS SQS, Kinesis, Amazon ML* and *DynamoDB*, or *Azure Event Hub, Stream Analytics, AzureML* and *CosmosDB*. (Gannon, Barga, and Sundaresan 2017, 17).

These two approaches are similar in that neither of them concern themselves with resource management. They are also frequently used together. For example, a FaaS function can rely on a BaaS database for state management. Roberts (2018) describes the two approaches as preferring "choreography over orchestration". I understand this phrase to mean that the control flow is born from interaction of the individual components, instead of keeping control in some centralized location. This means that each component is more architecturally aware, which is a common idea also with MSA applications (Roberts 2018).

Serverless architecture does have some drawbacks, some of which are inherent to the paradigm and some which can be at least partially mitigated with careful implementation (Roberts 2018). Inherent drawbacks cannot be entirely fixed, and need to be always considered. With BaaS and FaaS, you have to give up some control of your system to a third-party vendor. As features vary between vendors, you might suffer from vendor lock-in if migrating to another service would require significant changes to code, operational tooling, or even architecture. Because it is more economic, vendors might run multiple instances of software for several different customers on the same machine, possibly even on the same hosting application. This multitenancy might be unacceptable in strictly regulated domains. More vendors results in a larger surface area for cyber attacks, and more FaaS functions means that identity- and access management issues such as IAM policy errors are compounded. (Roberts 2018).

Roberts (2018) points out that implementation drawbacks can be mitigated by either your actions, or by the vendor improving their service. Immature platforms might offer less

options for configuration, monitoring, deployment, packaging, versioning, and debugging (Roberts 2018). Execution duration for FaaS functions is limited, which might require you to re-architect tasks that require long running times. It is entirely possible for you to run a denial-of-service (DoS) attack on yourself unwittingly if you accidentally or intentionally launch a large number of instances, because e.g. AWS has a limit for maximum number of running instances per cloud account. (Roberts 2018).

Testing can become a pain point. Sometimes a BaaS vendor might not allow load testing at all, or will at least bill you dearly (Roberts 2018). Setting up and tearing down state might be difficult with third-party services. FaaS approach will almost certainly lead to a high number of small units, and replicating the entire cloud environment on your local machine is impossible. This makes integration testing more important than with other architectural styles. (Roberts 2018).

Despite these drawbacks, serverless architecture brings many benefits to the table. Amazon Web Services (2023) advertise their serverless offerings to provide automatic scaling, built-in high availability, pay-for-value billing model, and built-in service integrations. It eliminates infrastructure management tasks, which increases agility and optimizes costs (Amazon Web Services 2023). Reduced operational and development costs make sense due to economies of scale and commodified services for common functionality (Roberts 2018). FaaS is especially economical for occasional or inconsistent traffic, because the FaaS service spins only the necessary number of instances up or down, and you often are billed down to microsecond accuracy. This way you don't need to overprovision a VM for peak traffic. For predictable and consistent traffic however a traditional VM might be cheaper, as then it is easier and safer to keep the computing hosts at high CPU utilization. Also, time-to-market can be considerably shortened because continuous iteration is easier due to simple and fast redeployment of individual components. (Roberts 2018).

2.3.4 Cloud architecture

A typical layered architecture might include separate layers for presentation, business logic, and data management (Pahl, Jamshidi, and Zimmermann 2018, 5). Even though the devel-

oper has no direct access to the hardware of the cloud platforms, cloud software is mapped onto physical tiers and their services. A physical tier means the physical location of the application execution, and the places where software layers are deployed and where they run. Each node in a cloud architecture should handle management and recovery tasks themself, which is why cloud services are self-managed entities. (Pahl, Jamshidi, and Zimmermann 2018, 5).

Due to these factors and the need for continuous services, Pahl, Jamshidi, and Zimmermann (2018, 5) think that the cloud requires its own architectural style. They define cloud architecture as "*an abstract model of a distributed cloud system with the appropriate elements to represent not only application components and their relationships but also the resources these components are deployed on and the respective management elements*". This aligns with my experiences, where architectural diagrams for cloud-native applications always seem to include the names of the used platform services. For example in case of an application built on AWS, the diagram will specify each usage of *SNS*, *SQS*, *Cognito*. *S3*, *Lambda*, *DynamoDB*, *API Gateway* and so forth, instead of using generic terminology.

Cloud-native architectural style described by Pahl, Jamshidi, and Zimmermann (2018, 5) includes a set of principles and patterns. Principles are service-orientation, virtualization, adaptation, and uncertainty. Service-orientation can be considered a group of three related principles: layering, modularity, and loose coupling. Layering will be necessary, because services will likely have different lifecycles, interface granularities, and so on. This will lead to the communication containing more indirections, which might require additional communications infrastructure. Modularity means refactoring an application into services and exposing only their interfaces, encapsulating implementation details. This makes integrations and composition easier, but requires a way to locate and call these smaller services. Care must also be taken to avoid service calls creating undesired side effects for state management. Because the separate services in a modular application are in some way dependent on each other, coupling should be minimized where possible. Loose coupling can be achieved e.g. with a messaging system, which decouples time, location, and even platform. Stateless services make loose coupling easier, as service calls to stateless services are less likely to result in side effects. (Pahl, Jamshidi, and Zimmermann 2018, 3–8).

Virtualization is generally considered a necessity to manage a shared pool of cloud resources effectively, with optimised cost benefits and economies of scale (Pahl, Jamshidi, and Zimmermann 2018, 8). CSPs offer services for virtualizing infrastructure such as VMs, and platforms such as container orchestration services. Emphasizing adaptation means that the service should meet its requirements with minimal human intervention despite changing circumstances such as system resources, errors, or usage patterns. Unfortunately, in the cloud it is difficult to map requirements to underlying infrastructure. Another thing to keep in mind is that placing a burden on the infrastructure will also map directly to the cloud cost model. This means that cost should be considered in adaptation requirement choices, excluding some as too costly. (Pahl, Jamshidi, and Zimmermann 2018, 3–10). In my view it is beneficial if an architect knows the cloud service provider's offerings almost to the infrastructure level, to the extent it is possible.

Uncertainty is always present in the cloud. It is caused by physical and logical distribution, heterogeneity, and multi-user involvement in changing contexts (Pahl, Jamshidi, and Zimmermann 2018, 3). It is also possible that monitoring data is unreliable or inadequate, if multiple distributed monitoring systems are in use. This makes reliable measurement of quality attributes of cloud software difficult. For example, creating or freeing cloud resources such as VMs is not immediate, which contributes to uncertainty during the change as the workload varies. (Pahl, Jamshidi, and Zimmermann 2018, 3–11).

2.4 Maintainability in cloud-native architecture

In this section I will tie together the topics I discussed in previous sections: architecture, maintainability, and cloud-nativity.

The Twelfe-Factor App (Wiggins 2017) is a methodology for building software for the web and software-as-a-service model (**SaaS**), and as such fits the cloud-native paradigm. It is intended to increase awareness of system-level problems in modern software development, offer a shared vocabulary to discuss them, and showcase a selection of broad concept-level solutions to them. Some of these solutions will support maintainability. Solution #2 encourages to explicitly declare and isolate dependencies. This means that one should never rely

on the supposed existence of system-wide packages or tools. If you need them, you should package them along with your application. All dependencies should be explicitly declared, and the environment should be isolated from external dependencies. These can be achieved by using the preferred tools in your programming language, such as *Gemfiles* and *bundle exec* with *Ruby* programming language, or *pip* and *virtualenv* with *Python*. (Wiggins 2017). Vettor, Pine, Jain, et al. (2022) concur that packaging dependencies in the container image of a microservice belonging to an MSA application fulfills this solution. This speeds up development, as a new developer will only need the codebase, the language runtime, and a dependency manager to run the application.

A ”log” is a stream of aggregated, time-ordered events collected from the output streams of all running processes and backing services (Wiggins 2017). Logs are often saved to a text file for later browsing. I consider software logs to be an inseparable part of a software maintainer's daily life. Twelve-factor app solution #11 says that logs should be treated as event streams, and forwarded unchanged into the standard output. This means relying on the platform's log management capabilities instead of trying to manage the log files yourself. Then the logs will be directed to the execution environment, which manages the logs for you. (Wiggins 2017).

In section 2.2.3 I mentioned ”smells” that are synonymous to antipattern, flaw, or anomaly (Mumtaz, Singh, and Blincoe 2021, 1). Service-oriented architecture was covered nicely in the research, but unfortunately there was only a little research into smells related to MSA or cloud architectures; the researchers identified only one research about cloud architecture smells. The researchers suggest more investigations into these architectural styles that are currently popular in the industry. Maintainability was the most commonly studied quality characteristic, which is fortunate for the researchers as this can shed light into the efforts required to fix the problems resulting from the smells. There is however still room for investigating the sub-characteristics of maintainability, such as testability and reusability. Modularity and modifiability were well covered in the available research. (Mumtaz, Singh, and Blincoe 2021, 21–22).

Almost like an answer to the suggestion by Mumtaz, Singh, and Blincoe (2021), Licht-

enthäler and Wirtz (2022) formulated a quality model[7] that is focused on the design time attributes of cloud-native software. This makes it possible to evaluate architecture before the implementation, because it doesn't rely on source code. They say that the model can be used to approximate the level of "cloud-nativeness" of the application. The higher-level quality aspects in the model are mapped from the ISO/IEC 25010 standard[8]. According to the ISO/IEC 25010 standard, the quality aspect maintainability can be split into sub-characteristics modularity, reusability, analysability, modifiability, and testability. Lichtenthäler and Wirtz (2022) add simplicity to this list. The model then connects these quality aspect to the so-called "product factors". Product factors characterize an architecture, and evaluating them will give us an idea how to practically improve the quality of the architecture in relation to that aspect. Additionally, the researchers list the metrics related to each product factor, but I omit them here for sake of brevity. (Lichtenthäler and Wirtz 2022).

The quality aspect reusability can be improved by focusing on standardization on system, component, and link levels (Lichtenthäler and Wirtz 2022). The goal is to have similarity between components. "System" means the cloud-native application as a whole. A "component" is an abstract entity for representing a distinct part of the system that provides certain functionalities. It can for example be a service or a certain cloud resource. A "link" then is a directed potential connection between a specific component and a specific endpoint of a different component. (Lichtenthäler and Wirtz 2022).

The quality aspect modularity is related to product factors service-orientation, isolated state, and loose coupling (Lichtenthäler and Wirtz 2022). Each of these three contains additional nested product factors. Service-orientation can be deconstructed into limited functional scope, and separation by gateways. Limited functional scope consists of nested product factors limited data scope, limited endpoint scope, and command query responsibility segregation. Isolated state means moving majority of the system state into specialized stateful services, and keeping most of the other services as stateless as possible. Loose coupling can be achieved by implementing asynchronous communication, and communication partner abstractions. Focusing on these factors would improve modularity as a whole. (Lichtenthäler

7. https://r0light.github.io/cna-quality-model/
8. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

and Wirtz 2022).

The quality aspect analysability is improved by implementing automated monitoring. This can mean consistent centralized logging and metrics, distributed tracing of invocations, and health and readiness checks (Lichtenthäler and Wirtz 2022). Monitorability is related to analysability, and is especially relevant for MSA systems, because such systems contain many dynamic structures and behaviors (Li et al. 2021, 13). It is possible to monitor many aspects of the stack separately: infrastructure, such as VM or a container metrics; software metrics such as response time; or the platform metrics, such as network latency. However with CNAs, we only need to worry about monitoring our own services, and leave the platform and infrastructure monitoring to the cloud service provider. In general, monitoring includes generating, storing, processing, and presenting the monitored data (Li et al. 2021, 13). These steps might require installing an agent and other additional tooling, but luckily cloud services often provide basic functionality out of the box.

The quality aspect simplicity can be broken down into sparsity, operation outsourcing, and usage of existing solutions for non-core capabilities. Operation outsourcing means using managed infrastructure, and managed backing services (Lichtenthäler and Wirtz 2022). It is understandable how this increases maintainability: the lower the scope of your maintenance responsibilities, the higher the maintainability. Cloud service providers offer backing services to facilitate building software. Backing services can help with monitoring, streaming, security, analytics, message brokering, storage, caching, and other needs (Vettor, Pine, Jain, et al. 2022). One should favor backing services provided by the cloud service provider in order save time and money by not having to worry about the backing service's performance, security or maintenance. Backing services are best utilized as an attached resource, that are dynamically attached to the application as an external configuration, for example via environment variables. This enables us to attach and detach a backing service without code changes. Another best practice is to implement a middle layer that sits between the business code and backing service. This helps avoid being tightly coupled to specific APIs, and changing the service will be easier if needed. (Vettor, Pine, Jain, et al. 2022).

The quality aspect modifiability can be improved by emphasizing service independence, which in turn can be achieved with low coupling, functional decentralization, limited request

trace scope, logical grouping, and backing service decentralization. Implementing abstractions for addressing these services will also benefit modifiability. Automated infrastructure provisioning is important, for which the natural companion is infrastructure-as-code (**IaC**). It helps automate platform provisioning and software deployment, because you are able to utilize software development practices such as testing and version control for the infrastructure definitions (Vettor, Pine, Jain, et al. 2022). IaC technologies include *AWS Cloudformation*, *Azure Resource Manager (ARM)*, *Azure Bicep*, *Terraform*, and *Ansible*. Resource names, locations, capacities and secrets are parametrized and dynamic in IaC. IaC scripts are also designed to be idempotent, meaning that you can change one resource's definition and rerun the script without altering the other resources. This makes IaC play nicely with continuous integration and continuous deployment (CI/CD) technologies: changes to infrastructure are easy to execute. If deployments are easy, integration and deployment likely happens more often than once a quarter, which leads to better communication and mistakes that are cheaper to fix, resulting in higher quality software. (Vettor, Pine, Jain, et al. 2022).

If a system is highly testable, it is able to demonstrate its flaws through testing (Li et al. 2021, 17). Testability is important with MSA applications, because microservices go through many changes during their life, and their interactions are often complex. By maximizing testability, we aim to minimize threats for performance, availability, and security, in addition to maintainability. Testability can be increased by API documentation and management, where we manage API versions and changes over time. This management should be automated as far as possible by tools such as *Swagger* and *RAML*, to avoid "forgetting" to update documentation. (Li et al. 2021, 17).

Testing can be made easier - and in turn more effective - by implementing automatic test procedures (Li et al. 2021, 17). Increasing automation in the testing process reduces the complexity and effort required. Comprehensive manual testing is in many cases impossible, due to the complexity inherent to MSAs. Automated testing procedures align well with the DevOps approach, which emphasizes automating away as much manual work as possible, in order to avoid tedium and human mistakes. Many different testing paradigms can be automated, not only unit tests. As a few examples, *LBTest* uses machine learning to automate end-to-end testing, *EVOMASTER* uses an evolutionary genetic algorithm for service testing,

Diffy uses a continuous delivery pipeline for regression testing, and *GMAT* uses graph-based dependency analysis for contract testing. Integrating these tools into the development process will require a learning curve, especially when the aimed at production software. (Li et al. 2021, 18).

This concludes the overview of the scientific background of maintainability in cloud-native architecture. In the following chapters I will present the basis of my own research: research method, results, and analysis. After that I will draw conclusions, and describe how they relate to the literary background I presented in this chapter 2.

3 Method

In this chapter I present the research questions, method, and the research process.

3.1 Research questions and choice of method

Through the research, I aimed to answer three research questions:

RQ1 How much importance do cloud architects place on maintainability during the design phase?

RQ2 How to address maintainability concerns when architecting cloud-native applications?

RQ3 Do the recommendations from literature match the views of architects working in the field?

A digital survey appeared to me to be the best-fitting research method for answering these questions. It probably would have been possible find existing data sets through a literary review, but finding a reasonable amount of existing data seemed unlikely especially for **RQ1**.

3.2 Survey

I targeted the survey at cloud architects of *Nordcloud, an IBM company*, where I work. When weighing my options how to collect the data necessary to answer the research questions, I decided to only utilize my current workplace as the target organization. The reasoning behind this was twofold: proximity through my current employment, and the fact that the company employs numerous cloud architects who had experience and, hopefully, opinions about their craft. The company uses the *Slack* tool for internal asynchronous messaging. I had access to the company's *Slack* workspace, which made contacting potential survey respondents convenient.

I used *Google Forms* for creating the survey and collecting the results. Before the respondent could proceed to answer the actual questions, the survey briefly described the research topic. There was also a data collection disclaimer with a mandatory agree-disagree choice. If the

respondent disagreed with the disclaimer, they were thanked for initial interest, and encouraged to submit the form nevertheless, so I would know that the terms were not preferable.

The first half of the survey gathered some demographic insights:

Q1 Years of experience in cloud-native architecture?
Q2 Years of experience in software architecture in general?
Q3 Years of experience in IT in general?

Each domain field mentioned in these questions was intended to be a subset of the next: in my view, IT is a superset of software architecture, which is a superset of cloud-native architecture. All three questions were multiple-choice, single answer with these ranges:

- 0-2 years
- 2-5 years
- 5-10 years
- 10+ years

Second half of the survey was dedicated to the main subject matter:

Q4 How do you prioritize these software quality attributes when designing cloud-native architecture?
Q5 How do you address maintainability with platform- and technology choices?
Q6 How do you address maintainability with application architecture?

Survey question **Q4** is directly linked to research question **RQ1**. The question listed a predefined selection of software quality attributes:

- Performance
- Reliability
- Maintainability
- Scalability
- Security

The respondent needed to set the above quality attributes in a priority order. Available options were:

- Most important
- More important
- Important
- Less important
- Least important

Survey questions **Q5** and **Q6** are both linked to research questions **RQ2** and **RQ3**. The input was a free text field, so that the respondents could express their thoughts in a non-structured way. I did not want to limit the answer space with my own lack of knowledge by providing a predefined set of options. My intention was to later interpret the answers in a qualitative fashion to find common factors between the answers.

On Monday 20.03.2023 I posted an introductory message along with a link to the survey to the company's internal *Slack* channels in order to reach most of their cloud architects. These channels were used:

- #tech-infra
- #aws
- #azure
- #google-cloud

Some of these communities had overlapping audience, but the total reach was over a thousand people. I assumed the company to employ a few dozen cloud architects, but I didn't have exact numbers. My goal was to get 10-20 answers, to conduct any meaningful statistical analysis on the data. I was prepared to repost the survey in *Slack* once a week up to four times until I decide I have enough answers to proceed. In the end there was no need to repost, as a sufficient number of architects responded after the initial posting. In early April I managed to convince one additional architect in-person at the company's Jyväskylä office to respond. Some stray responses were recorded up to 21.4.2023, with a final tally of 15 responses.

I could have widened the target audience to other companies, if I wanted to collect a larger sample base. I decided against it, because my preliminary plans at the time did not require more than 20 samples. I planned to do some amount of qualitative processing on the results,

in addition to basic quantitative analysis. If I had included other companies in the research, I would have likely approached the companies via email instead.

I used my own university-provided *Google* account for all survey-related functions. Individual answers and their aggregated analysis will be stored in *Google Forms* and *Google Sheets* in this student account until the end of my study rights and subsequent removal of the account.

All answers were collected and analyzed anonymously. Email addresses or other pieces of personal information were not collected. Responding to the survey required logging in with a *Google* account, but it was only enforced to kindly remind the busy architects if they had already responded to the survey.

3.3 Analysis

My approach to the analysis of the survey results was not strictly quantitative or qualitative, but rather a mix of both depending on the source data. Questions **Q1** to **Q4** had a limited input space with multiple-choice fields, and as such were a natural candidate for basic quantitative processing. To answer research question **RQ1**, I decided to calculate the priority of maintainability relative to the other quality attributes based on answers to question **Q4**.

The free text questions **Q5** and **Q6** benefitted from a more qualitative approach. Because the responses were in an unstructured format, I needed multiple steps to transform the data to a format that could be aggregated and analyzed as an answer to research questions **RQ2** and **RQ3**:

- Interpret and list the suggestions from answers to questions Q5 and Q6.
- Normalize the suggestions to find common categories.
- Find the most popular categories and point out possible outliers.
- Compare the categories to solutions proposed in literature.

Because I gathered some demographic data with survey questions **Q1** to **Q3**, I could formulate additional insights by combining those results with results to the domain questions **Q4** to **Q6**. It is possible to calculate statistical correlation between years of experience and:

- Perceived importance of maintainability.
- Number of categories proposed in Q5 and Q6.
- Popularity of categories proposed in Q5 and Q6.

These last items are not the main focus of the thesis, but could be interesting as additional sidenotes. **remove mention of these last analysis items, if you don't execute them**

4 Results

In this section I present the results I gathered from the survey.

4.1 Demographics (Q1-Q3)

Table 1 shows respondent experience in cloud-native architecture specifically. Table 2 shows respondent experience in software architecture in general. Table 3 shows respondent experience in information technology in general.

Table 1. Respondent experience in cloud-native architecture, $n = 15$

Option	Responses	Percentage	Rank
0-2 years	2	13.33%	3
2-5 years	6	40.00%	2
5-10 years	7	46.67%	1
10+ years	0	0.00%	4

Table 2. Respondent experience in software architecture in general, $n = 15$

Option	Responses	Percentage	Rank
0-2 years	3	20.00%	3
2-5 years	4	26.67%	2
5-10 years	3	20.00%	3
10+ years	5	33.33%	1

Table 3. Respondent experience in information technology in general, $n = 15$

Option	Responses	Percentage	Rank
0-2 years	0	0.00%	4
2-5 years	1	6.67%	3
5-10 years	3	20.00%	2
10+ years	11	73.33%	1

4.2 Prioritization of maintainability (Q4)

To calculate a priority score for each quality attribute, I gave the options an incremental integer value from 1 to 5. The value 1 corresponds to option *Least Important*, while value 5 corresponds to *Most important*. The exact values chosen here are not of importance as long as the difference between each consecutive option is equal, to avoid introducing additional bias. This makes our scale analogous to the Likert scale (Likert 1932), which is often used for questionnaires in research. Table 4 lists the integer values given to each priority option, and table 5 portrays relevant metrics calculated for each quality attribute.

Table 4. Priority options and their integer values

Priority	**Value**
Most important	5
More important	4
Important	3
Less important	2
Least important	1

Table 5. Quality attributes, their average priorities, and other metrics, $n = 15$

QA	**Average**	**Min**	**Mode**	**Max**
Security	4.07	1	5	5
Reliability	3.80	1	5	5
Maintainability	2.67	1	2	4
Scalability	2.27	1	1	5
Performance	2.20	1	1	4

4.3 Ways to improve maintainability (Q5 & Q6)

I intended the two questions **Q5** and **Q6** to be strictly scoped to two separate areas of tech selections and architecture. Instead, the responses were varied and included suggestions touching multiple different areas. I think this was luck on my part, because otherwise I would not have received process- or philosophy-related answers.

I categorized the results to three top-level categories, two of which align to the survey questions **Q5** and **Q6**:

- Platform and technology choices
- Application architecture
- Others (processes, best practices, philosophy, ...)

For each top-level category, I further interpreted subcategories from each suggestion, and aggregated them where reasonable. For example, I consider all the following responses belonging to subcategory ”common technologies”:

- ”Use up-to-date (but established) frameworks” (respondent 4)
- ”Future-proof technologies” (respondent 2)
- ”By using commonly used technologies” (respondent 13)

Redo the below list into 3 tables instead? –> category name, number of occurences, list of respondents... and fix it with latest categorizations from the google sheet

- Platform and technology choices
  - Common technologies
  - Managed services
  - Infrastructure as Code (IaC)
  - Configuration as Code (CaC)
  - Continuous Integration and Continuous Deployment (CI/CD)
- Application architecture
  - Microservices Architecture (MSA)
  - Statelessness
  - Automatic scaling
  - Modularity
  - Loose coupling
  - SOLID principles
- Others (processes, philosophy, etc)
  - DevOps
  - Clear naming of resources

  - Documentation
  - Code reviews
  - KISS principle

4.4 Experience vs. prioritization of maintainability

should be simple

4.5 Experience vs. suggested ways to improve maintainability

Slightly more involved. Which metrics to select?

- **number of suggestions?**
- **popularity of suggestions in context of survey?**
- **popularity of suggestions in the wider context of existing research?**

5 Discussion

In this section I further discuss the results presented in previous chapter. *I also consider existing literature and contrast my results against findings of those who have researched the topic before me.*

In table 6 are listed the distances between the average priorities. Distances are calculated from results shown in table 5. From these values we can draw multiple conclusions. The top two quality attributes, Security and Reliability, are unequivocally the most highly prioritized attributes in the minds of our cloud architects. The distance between second and third most prioritized attributes is by far the largest individual jump in values. Maintainability is sitting quite comfortable in the middle of the range. This likely means it is considered important, but not a priority if sacrifices need to be made to software quality. Performance and scalability are considered least important items to an almost identical degree.

Table 6. Average priorities and delta to next highest item

QA	Average	Δ
Security	4.07	-
Reliability	3.80	0.27
Maintainability	2.67	1.13
Scalability	2.27	0.40
Performance	2.20	0.07

—

seuraava kopsattu teoriaosiosta. Katso myös muut jutut osiosta "how to improve maintainability"

Maintainability can be improved in the architectural design phase by emphasizing encapsulation, modularity, and scalability (ISO/IEC/IEEE 12207:2017(E) 2017). Documenting the architecture and system reduces the efforts required to reverse-engineer the system and components when a fix is needed. The decisions made during architecting can also affect the need and possibilities for e.g. remote diagnostics and maintenance, roll back, backup, and

recovering data. (ISO/IEC/IEEE 12207:2017(E) 2017, 95–96).

Summary: Advantages and challenges of cloud-nativity

To sum up, cloud-nativity offers many advantages over on-premises software. Cloud-native approach allows the wielder to more easily respond to changing market situations with speed and confidence (Vettor, Pine, Jain, et al. 2022). Parts of a CNA can scale independently based on demand, which pairs well with elasticity and the ”pay what you use” philosophy of public cloud: there is no need to overprovision infrastructure for peak consumption, which in turn helps manage costs (Patrizio 2018). CNAs have high support for automation, which enables the developers to concentrate on relevant challenges instead of tedium (Patrizio 2018). Combined with the ability to deploy small individual services instead of an entire complex system, a CNA is easy to update. This boosts productivity when development teams can make both small-scale and sweeping changes with predicable results, and deliver new features to customers more often (Noval et al. 2022; Patrizio 2018).

When following cloud-native principles, the application will be loosely coupled, resilient, manageable and observable (Noval et al. 2022). Another staple of such software is statelessness, which makes it easier to scale to multiple servers, leverage caching, use less storage space, and avoid vendor lock-in by not being reliant on any specific type of server (Patrizio 2018). You can avoid much downtime because CSP backing systems have inbuilt redundancy, and are geographically distributed (Patrizio 2018).

In previous chapters I have written about challenges relating to a specific way of implementing CNAs. Patrizio (2018) point out some challenges related to cloud-native on an organization level. A lift-and-shift operation, where a monolithic on-premises application is simply transplanted onto a cloud platform as-is, does not benefit from the cloud at an architectural level. On the other hand, a full rewrite is an undeniable engineering challenge. The cloud-native approach requires organizations to shift their mindset towards more agile principles, such as Minimum Viable Product (MVP) development, automation, multivariate testing, fast iteration, observability, and DevOps. There is also a need to come to terms with the cloud-centered security model, and how to handle costs incurred from operating the cloud environments. Lastly, it might be difficult to find enough competent personnel to implement

all the previous points in addition to the organization's other goals. (Patrizio 2018).

6 Conclusions

Bibliography

Amazon Web Services. 2023. “Serverless Computing”. Visited on September 2, 2023. https://aws.amazon.com/serverless/.

Arvanitou, Elvira Maria, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Matthias Galster, and Paris Avgeriou. 2017. “A mapping study on design-time quality attributes and metrics”. *Journal of Systems and Software* 127:52–77. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2017.01.026.

Bass, Len, Paul Clements, and Rick Kazman. 1998. *Software Architecture in Practice.* USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201199300.

———. 2003. *Software Architecture in Practice.* SEI series in software engineering. Addison-Wesley. ISBN: 9780321154958. http://books.google.fi/books?id=mdiIu8Kk1WMC.

Bogner, Justus, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. 2018. “Limiting Technical Debt with Maintainability Assurance - An Industry Survey on Used Techniques and Differences with Service-and Microservice-Based Systems”. In *1st International Conference on Technical Debt (TechDebt’18).* https://doi.org/10.1145/3194164.3194166.

Bouwers, Eric, and Arie van Deursen. 2010. “A Lightweight Sanity Check for Implemented Architectures”. *IEEE Software* 27 (4): 44–50. ISSN: 1937-4194. https://doi.org/10.1109/MS.2010.60.

Broy, Manfred, Florian Deissenboeck, and Markus Pizka. 2006. “Demystifying Maintainability”. In *Proceedings of the 2006 International Workshop on Software Quality,* 21–26. WoSQ ’06. Shanghai, China: Association for Computing Machinery. ISBN: 1595933999. https://doi.org/10.1145/1137702.1137708.

Gannon, Dennis, Roger Barga, and Neel Sundaresan. 2017. “Cloud-Native Applications”. *IEEE Cloud Computing* 4 (5): 16–21. ISSN: 2325-6095. https://doi.org/10.1109/MCC.2017.4250939.

“Google Scholar”. 2023. Visited on February 26, 2023. http://scholar.google.com.

Gorla, Narasimhaiah, and Shang-Che Lin. 2010. “Determinants of software quality: A survey of information systems project managers”. *Information and Software Technology* 52 (6): 602–610. ISSN: 0950-5849. https://doi.org/https://doi.org/10.1016/j.infsof.2009.11.012.

ISO/IEC 5055:2021(E). 2021. *Information technology - Software measurement - Software quality measurement - Automated source code quality measures.* Standard. Geneva, CH: International Organization for Standardization.

ISO/IEC/IEEE 12207:2017(E). 2017. *Systems and software engineering – Software life cycle processes.* Standard. International Organization for Standardization. https://doi.org/10.1109/IEEESTD.2017.8100771.

ISO/IEC/IEEE 42010:2007. 2007. *Recommended Practice for Architectural Description of Software-Intensive Systems.* Standard. International Organization for Standardization. https://doi.org/10.1109/IEEESTD.2007.386501.

“JYKDOK international articles search”. 2023. Visited on February 26, 2023. https://jyu.finna.fi/primo.

Kaihlavirta, Johanna. 2022. “Time tracking in software maintenance service”. Master’s thesis, University of Jyväskylä. https://jyx.jyu.fi/bitstream/handle/123456789/84170/URN%3aNBN%3afi%3ajyu-202212015437.pdf?sequence=1&isAllowed=y.

Koskimies, K., and T. Mikkonen. 2005. *Ohjelmistoarkkitehtuurit.* Valikko-sarja. Talentum. ISBN: 952-14-0862-6.

Kratzke, Nane, and Peter-Christian Quint. 2017. “Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study”. *Journal of Systems and Software* 126:1–16. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2017.01.001.

Li, Shanshan, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. 2021. “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review”. *Information and Software Technology* 131:106449. ISSN: 0950-5849. https://doi.org/https://doi.org/10.1016/j.infsof.2020.106449.

Lichtenthäler, Robin, and Guido Wirtz. 2022. “Towards a Quality Model for Cloud-native Applications”. In *Service-Oriented and Cloud Computing,* edited by Fabrizio Montesi, George Angelos Papadopoulos, and Wolf Zimmermann, 109–117. Cham: Springer International Publishing.

Likert, Rensis. 1932. “A Technique for the Measurement of Attitudes”. *Archives of Psychology* 140:1–55.

Moses, John. 2009. “Should we try to measure software quality attributes directly?” *Software Quality Journal* 17 (2): 203–213. ISSN: 1573-1367. https://doi.org/https://doi.org/10.1007/s11219-008-9071-6.

Mumtaz, Haris, Paramvir Singh, and Kelly Blincoe. 2021. “A systematic mapping study on architectural smells detection”. *Journal of Systems and Software* 173:110885. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2020.110885.

Noval, Ahmad, Chris Aniszczyk, Vincent Rabah, Orlin Vasilev, Michal Jakobczyk, Mario Cisterna, Dominik Liebler, et al. 2022. “CNCF Cloud Native Definition v1.0 · cncf/toc”. Visited on April 10, 2023.

Pahl, Claus, Pooyan Jamshidi, and Olaf Zimmermann. 2018. “Architectural Principles for Cloud Software”. *ACM Trans. Internet Technol.* (New York, NY, USA) 18 (2). ISSN: 1533-5399.

Patrizio, Andy. 2018. “What is cloud-native? The modern way to develop software”. *InfoWorld.com* (San Mateo), https://www.proquest.com/trade-journals/what-is-cloud-native-modern-way-develop-software/docview/2055357601/se-2?accountid=11774.

Roberts, Mike. 2018. “Serverless Architectures”. Visited on September 2, 2023. https://martinfowler.com/articles/serverless.html.

Rudisail, Brad. 2021. “Cloud Scalability vs Cloud Elasticity: Here’s How They Differ - Spiceworks”. Visited on April 10, 2023.

Vale, Guilherme, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. 2022. “Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs”. (Ithaca), https://www.proquest.com/working-papers/designing-microservice-systems-using-patterns/docview/2619059217/se-2.

Vettor, Rob, David Pine, David Coulter, Nick Schonning, and Maira Wenzel. 2022. “Cloud-native communication patterns”. Visited on March 25, 2023.

Vettor, Rob, David Pine, David Coulter, Youssef Victor, Nick Schonning, and Maira Wenzel. 2022. “Cloud-native communication patterns”. Visited on March 25, 2023.

Vettor, Rob, David Pine, Nick Coulter, Maira Wenzel, and Steve Smith. 2022. “Cloud-native communication patterns”. Visited on March 25, 2023.

Vettor, Rob, David Pine, Tarun Jain, Szanto Peter, Omair Majid, Genevieve Warren, Kent Sharkey, et al. 2022. “What is Cloud Native?” Visited on March 25, 2023.

Wiggins, Adam. 2011. “The New Heroku (Part 4 of 4): Erosion-resistance & Explicit Contracts”. Visited on March 26, 2023. https://blog.heroku.com/the_new_heroku_4_erosion_resistance_explicit_contracts.

———. 2017. “The Twelve-Factor App”. Visited on March 26, 2023. https://12factor.net/.

remove the nocite command from end of thesis.tex before submitting!

Appendices