Marcel Schuchmann

Designing a cloud architecture for an application with many users

Master's thesis of mathematical information technology

April 29, 2018

University of Jyväskylä

Department of Mathematical Information Technology

Author: Marcel Schuchmann

Contact information: marcel.schuchmann@gmail.com

Supervisors: Oleksiy Khriyenko, Vagan Terziyan and Jyri Leinonen

Title: Designing a cloud architecture for an application with many users

Työn nimi: Pilviarkkitehtuurin suunnittelu sovellukselle, jolla on paljon käyttäjiä

Project: Master's thesis

Study line: Web Intelligence and Service Engineering

Page count: 73

Abstract: The aim of the thesis is to provide a guideline on how to design and implement a cloud architecture solution for an application with many users. For this, general cloud architecture approaches are presented. The theory part is based on techniques of designing a cloud architecture, cloud computing in general, virtualization, databases, and related work of comparisons of cloud computing services. The case objectives of a mobile payment application are stated and defined. On these objectives, a study is conducted on different kinds of cloud backend architecture solutions, which are the tier-based architecture, the message queue architecture, the microservice architecture and the Serverless architecture. The microservice architecture and the Serverless architecture are assessed to be the most promising architectures for the case, because of their excellent scalability. The microservice architecture in Amazon Web Services and the Serverless architecture in Firebase are practically implemented for the case and compared to each other. The Serverless architecture in Firebase is easy to implement and therefore an excellent decision for a cloud architecture with certain limitations. However, the microservice architecture is a more complex architecture, which should be considered if user limits are reached or more configuration possibilities in the architecture are needed.

Keywords: cloud, architecture, design, scalability, availability, reliability, n-tier, multi-tier, IaaS, virtualization, VM, message queue, microservice, PaaS, Docker, Serverless, functions, FaaS, Firebase, Realtime Database, AWS, ECS, Fargate, DynamoDB, Express, REST

Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
BASE	Basically Available, Soft state, Eventually consistent
CAP	Consistency, Availability, and tolerance to network Partitions
CLI	Command Line Interface
CRUD	Create, Read, Update, and Delete
ECR	AWS Elastic Registry Service
ECS	AWS Elastic Container Service
FaaS	Functions as a Service
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
JWT	JSON Web Token
NIST	National Institute of Standards and Technology
PaaS	Platform as a Service
REST	Representative State Transfer
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SQL	Structured Query Language

TCP	Transmission Control Protocol
VM	Virtual Machine
VPC	Virtual Private Cloud

List of Figures

List of Tables

Contents

1 Introduction

Globally there are about 4.6 billion mobile broadband subscriptions (Heuveldop, 2017). This could lead to a world where almost everyone will have a smartphone that is connected to the Internet. New trends and phenomena are being distributed more rapidly via recommendations over the Internet. IT companies, that are developing applications, could face an unexpected user rise. A good example for this is the mobile augmented reality game Pokemon Go, where the servers could not handle the massive demand of the users in July 2016 as the application was launched in North America and Australia. The actual user traffic of the game was ten times higher than the "worst-case" estimated traffic (Stone, 2016). For this game, the demand stayed high despite the server problems, but for another mobile application this could mean the end of the application. This leads to the main research question in this thesis; how should a cloud architecture be designed to handle a lot of different users of an application at the same time?

One of the rising trends, where many users are expected, is mobile payment, which can make paying faster and more convenient than using a credit card or cash. Different authors have identified that trust can lead to that more consumers accept and use mobile payment services (Y. Lu, Yang, Chau, & Cao, 2011; Schierz, Schilke, & Wirtz, 2010). Hence, for the case of a mobile payment application, it is essential to have a functioning system and to work as the user expects. In this study, a cloud architecture will be planned and analyzed for the case of a mobile payment application. The mobile payment application is developed in a Finnish start-up company called Sweetlakes Oy.

A mobile payment application can be classified under a domain of modern applications, which have many users, a rapidly changing number of users, and many small requests or transactions for the backend logic in a cloud. In this domain, a backend must be an available, scalable, and reliable service to respond fast and correctly to every request. In contrast, heavy processing tasks or storage of big amounts of data are not included in this domain. In this thesis, the term of many users is used for an indefinite number of users. Currently, many users in an application can refer to millions of active users per month. However, there exists

already social applications with billions of users, for example the WhatsApp messenger or the Facebook messenger (Constine, 2017; Sparks, 2017). Therefore, in future, as technology develops, and user amounts rise, many users could refer to billions of users.

1.1 Research question and objectives

The thesis answers the research question of how to design a cloud architecture for an application with many users. Therefore, different architecture approaches are presented, assessed, and compared to each other. Different objectives must be introduced and defined to assess the architectural solutions. Different actors of an application have different objectives towards the application. A user expects for example the application to function consistently and with a good performance. A developer or company wants an easy to develop architectural solution with low costs. From those expectations, the objectives of availability, scalability, reliability, and a low amount of needed resources for the application can be retrieved.

Availability is the proportion of time of a service in a working and reachable state (Toeroe & Tam, 2012). A 100 % availability for each service of the backend application logic of an application is desirable. The reasons for an unavailable service could be a failure of a component, a general outage of the cloud, an overload of the service or a bug in the system. A cloud vendor can provide assurances on the availability of a cloud with a Service Level Agreement (SLA) (Marston, Li, Bandyopadhyay, Zhang, & Ghalsasi, 2011), where a concrete availability is defined for each service and the compensation if the promised availability has not been provided. A common approach to improve the availability is the reduction of single point of failures in an architecture.

Scalability is the ability of a system to provide the correct amount of resources depending on the load (Bondi, 2000). In the case of the payment application, the cloud architecture should handle one payment per second as well as one thousand payments per second ideally with the correct amount of required resources. The provisioned resources correspond to the costs, which a cloud consumer pays the cloud provider. In the cloud computing field, it has been identified that it is a problem to scale up and down correctly with different user peak loads and therefore not to waste any resources (Armbrust et al., 2010). A cloud consumer

pays unnecessarily more for overprovisioning resources, which are not needed for a load. Hence, a cloud architecture should have a low scaling latency to adjust correctly to rapidly changing loads. Furthermore, the limit of scalability of an architecture can be measured with the amount of possible concurrent connections to a cloud service.

Reliability of an application consist of different user expectations regarding the application. Firstly, a user expects an application to function in a reliable way. This includes the consistency of the data and that transactions are correctly processed. Furthermore, the service should be available and have a good performance. Therefore, the performance of an application should not exceed a certain time. This is especially important for the case of the mobile payment application, which is advertised as a faster and more convenient payment method. Furthermore, reliability reflects the trust between a user and an application. A user trusts in the application to secure his personal data and not to misuse it in any way. Personal data in the case of the payment application are the payment credentials, which the user provides only if the user trusts in the application that his personal data is secured.

The needed resources for a cloud architecture can be divided in the amount of workload for developing and maintaining a cloud solution and the recurring monthly costs for using cloud resources. The amount of workload consists of planning and setting up a system. Furthermore, in the development process, the simplicity of deployment is important to reduce the work time for a developer. In general, the usage of cloud resources is paid monthly without any one-time payments. Especially for the case of the mobile payment application, it is important to have a low cost per payment, as well as to have a positive margin per payment. However, as well for any other application in the problem domain it is critical to reduce the recurring costs per month.

In the design process of the architecture, decisions and assessments are made by considering these objectives of availability, scalability, reliability and needed resources.

1.2 Research method

The thesis is a system development research presented by Nunamaker, Chen and Purdin (1990). Although the system development research is about 20 years old, it can be applied

to the research problem as it is a practical research and it has appropriate stages for decision making between different solutions. A system development research consists of 5 different stages, which are depicted in Figure 1. At first, a research problem is to be identified and the corresponding theory is presented. In the second stage the system architecture, which is going to be developed, is designed. For this the objectives, constraints and requirements must be identified and defined (Nunamaker Jr et al., 1990). The third stage is the presentation of alternative solutions and a decision is made between these solutions. In the fourth stage different chosen solutions are developed. At last, the developed systems are evaluated and compared on the objectives. The research process is an iterative process to improve the result continually.

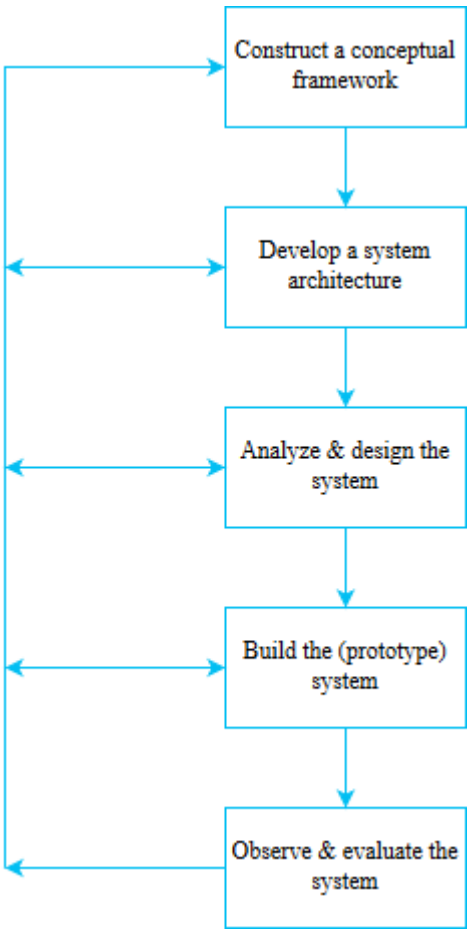

Figure 1 System development research process (Adapted from Nunamaker Jr et al., 1990)

1.3 Thesis structure

The structure of this thesis is as follows: at first, key concepts of cloud computing will be defined, and related work will be studied. Then the application architecture is introduced. After that, different solutions of cloud architectures will be presented and assessed on the objectives of the case. The solutions will be compared, and an architectural solution will be chosen. This solution will be implemented and compared to a modern Serverless approach implemented in Google Firebase, which the company of the case has initially chosen for their product. Then there will be a discussion about the best solution for this case and in a general way for applications in the same domain. After that, possible future work in this area will be presented. Finally, the results of this thesis are summarized in the conclusion part.

2 Designing a cloud architecture

Fowler describes that designing an architecture consists of two common elements, which are dividing a system into different parts and to make decisions that are hard to change in later stage of a system development process (2002). Hence, the design stage of an architecture should be made carefully and in detail. In this chapter, general concepts, and terms of designing a cloud backend architecture are defined and explained as cloud computing, virtualization, and databases. Furthermore, related work to the thesis is studied and discussed.

2.1 Cloud computing

A cloud backend is built on the technology of cloud computing. Mell & Grance of the National Institute of Standards and Technology (NIST) defining cloud computing (2011) as a

> *"model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

The model of cloud computing has five essential characteristics according to the NIST definition (Mell & Grance, 2011).

- *On-demand self-service.* The used service is automatically provisioned to the consumer on-demand.
- *Broad network access.* Clients can access the different services provided by the cloud through a standardized network.
- *Resource pooling.* Same resources of the cloud provider are consumed by different consumers on-demand.
- *Rapid elasticity.* Resources of the cloud can be extended rapidly by the cloud provider and for the consumer it feels that (s)he can use infinite resources.

- *Measured service.* The usage of cloud services is measured by the cloud provider and reported to the cloud consumer.

There are different approaches on how to deploy the backend architecture of an application to the cloud. The backend can be deployed to a public, private, community, or hybrid cloud as stated by the NIST definition (Mell & Grance, 2011).

- *Public cloud*. The cloud infrastructure is provisioned for the public use. A business, an educational institution, or a government can provide it.

- *Private cloud*. The cloud infrastructure is provisioned for exclusive use of an organization. The organization itself or a third party can provide it.

- *Community cloud*. The cloud infrastructure is provisioned for exclusive use of a community of consumers that have a shared concern. One or more organizations of the community or a third party can provide it.

- *Hybrid cloud*. The cloud infrastructure is a mix of unique entities of the other distinct cloud infrastructures.

In this study only a public cloud, which is provided and maintained by a public cloud vendor, is considered. Public cloud solutions are preferable for small companies compared to acquiring the hardware themselves, because there is not a high price of buying the hardware and one pays for what one uses (Armbrust et al., 2010). Furthermore, the deployment should happen on a public cloud in order to have a high scalability provided by the cloud vendor (Rountree & Castrillo, 2013).

Layering is a commonly used technique for a system designer to divide a system into smaller parts, so called layers (Fowler, 2002). Layers can be stacked vertically, where in a strict layered model a higher layer has only access to a layer below it, but a lower layer has no access to a higher layer (Brown et al., 2003; Fowler, 2002). This technique is used to describe models or architectures (Brown et al., 2003). Cloud computing is described as a 3-layered service model by the NIST definition (Mell & Grance, 2011). The model is illustrated in Figure 2.

- *Infrastructure as a Service (IaaS)*. Usage of physical hardware as storage, network, processing, and computing resources, which are managed and controlled by the cloud provider (Mell & Grance, 2011). The cloud consumer handles deployment and configuration of arbitrary software including operating systems (Mell & Grance, 2011). IaaS is also known as a virtualization layer, where computing resources are provided as a virtual machine (VM) (Zhang, Cheng, & Boutaba, 2010).

- *Platform as a Service (PaaS)*. The deployment and configuration of applications created by the cloud consumer with compatible programming languages, libraries and services on the cloud infrastructure, which is controlled and managed by the cloud provider (Mell & Grance, 2011).

- *Software as a Service (SaaS)*. Usage and controlling of user-related settings of applications provided by the cloud vendor. Clients can access the applications through a web browser, thin client interface or program interface. (Mell & Grance, 2011)

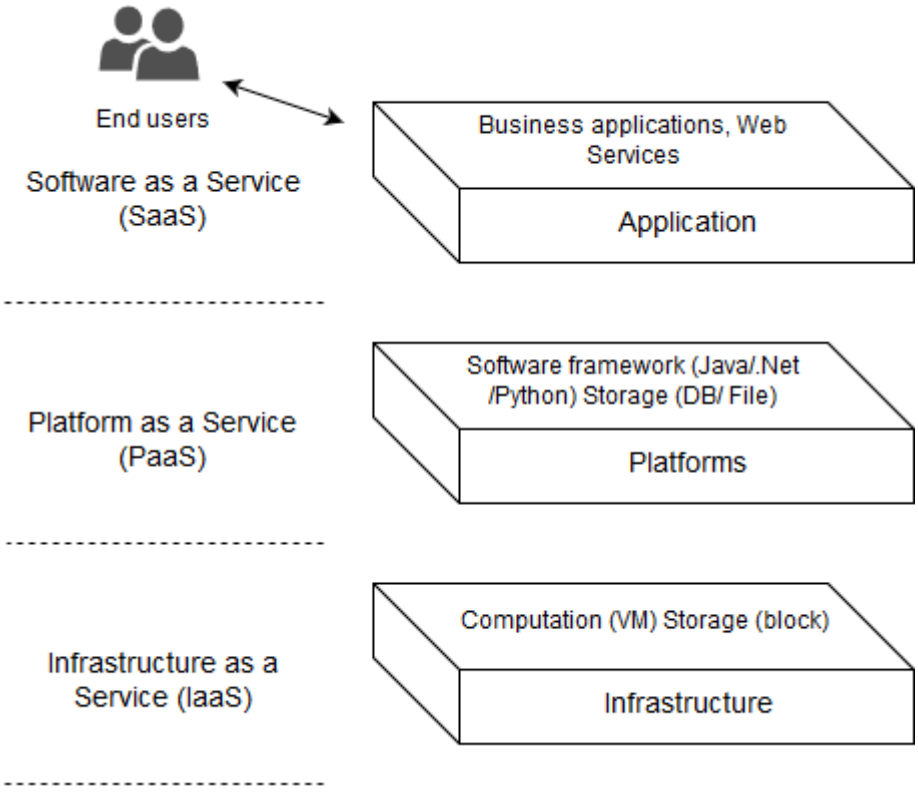

Figure 2. Cloud computing architecture (Adapted from Zhang et al., 2010)

Other definitions define cloud computing as a 2-layered service model (Armbrust et al., 2010) consisting of high- and low-level layers, where IaaS and PaaS are defined as one layer. For better differentiation in this thesis, they need to be distinct, because different architectural solutions are in these different layers. Furthermore, cloud computing has been defined as a 4-layered service model (Zhang et al., 2010), where the hardware in datacenters is considered a separate layer. For this thesis, it is not necessary to have a separate hardware layer, as the architecture design is independent of the used hardware and handled by the cloud provider. Additionally, the traditional service models can be extended with more specific service models, for example, the modern Serverless service models of Backend as a Service (BaaS) and Functions as a Service (FaaS), which are situated between SaaS and PaaS (Wolf, 2018). BaaS is the usage of third-party backend services directly from a client (Roberts, 2016). BaaS is especially designed for the mobile market with services like user management, push notifications and social media integration (Sareen, 2013). FaaS is a stateless function that consist of custom code that runs on a small compute instance managed by the cloud provider (Roberts, 2016). A function is executed by different events, which a client can trigger (Roberts, 2016; Wolf, 2018). Cloud consumers of Serverless service models share the same service installations and resources (Wolf, 2018).

2.2 Virtualization

Virtualization in cloud computing refers to the abstraction of a single hardware resource to multiple virtual resources, which are sharing the same hardware resource (Kusic, Kephart, Hanson, Kandasamy, & Jiang, 2009; Younge et al., 2011). Public cloud providers leverage the virtualization technology for a better hardware resource utilization in their clouds (Joy, 2015), because resources as CPU, memory or disk space are dynamically provisioned to the cloud consumers on demand (Kusic et al., 2009).

2.2.1 Virtual machine (VM)

A virtual machine is a simulated machine with its own isolated operating system, where several applications could run (Kusic et al., 2009; Xavier et al., 2013). A virtual machine

works on top of a virtual machine monitor, which is also called hypervisor (Younge et al., 2011). A hypervisor runs different kernels on top of the hardware and organizes the hardware provisioning to the different VMs (Joy, 2015).

2.2.2 Container technology

A lightweight alternative to the usage of a hypervisor is a container-based virtualization (Xavier et al., 2013). Containers work at the operating system level, therefore they are sharing the same operating system host kernel efficiently (Joy, 2015; Xavier et al., 2013). More containers can run on a single host compared to VMs, because containers do not run a full operation system, which makes them require fewer resources (Joy, 2015).

2.3 Database

One part of cloud computing is the storage of data in a database. In an application with many users, a database must be able to store the data of each user. Furthermore, the database must be able to handle a lot of concurrent operations on it (J. Han, Haihong, Le, & Du, 2011). Brewer has stated the CAP theorem for shared-data systems, that only two of three properties can be fulfilled of Consistency, Availability, and tolerance to network Partitions (2000). In current applications and distributed data systems, the tolerance to network partitions is needed and a system designer must decide between consistency and availability (Okman, Gal-Oz, Gonen, Gudes, & Abramov, 2011). In a database, the most common operations are Create, Read, Update, and Delete (CRUD) an entry.

A database in the cloud can be self-maintained on IaaS or PaaS, which would mean an increased workload for the enterprise or can be used as a service provided by the cloud vendor. A database as a service can be among others, a relational database (Curino et al., 2011) or a NoSQL database (J. Han et al., 2011). Important for all database concepts is that user-related data can be only accessed after a user authentication.

2.3.1 Relational database

A relational database stores data in structured tables, where different categories are described as columns and entities as rows (Leavitt, 2010). Entities are identified by keys and can have relations to other entities. A relational database ensures the characteristics of Atomicity, Consistency, Isolation, and Durability (ACID) for transactions (Pokorny, 2013). Mostly the Structured Query Language (SQL) is used for querying and updating a relational database. A relational database offers a large feature set with SQL, which increases the complexity and might not be needed in every case (Leavitt, 2010).

Traditionally a relational database is located on one server, which makes it difficult to scale a relational database in a distributed way (Leavitt, 2010). Recently, relational databases solutions have improved their scalability by distributing data over several server nodes in a "shared nothing" architecture (Cattell, 2011). The server nodes, also called shards, are replicated in clusters to support a recovery of data (Cattell, 2011).

2.3.2 NoSQL database

NoSQL databases have been introduced to overcome the downfalls of a complex relational database. NoSQL means "not only"-SQL and is the representative name of all modern non-relational datastores (Leavitt, 2010; Pokorny, 2013).

NoSQL databases can be divided into key values store, column-oriented database, document-based stores or graph database (J. Han et al., 2011; Leavitt, 2010; Tauro, Aravindh, & Shreeharsha, 2012).

- *Key value store*. An indexed key retrieves values. A key value store can be structured or unstructured.
- *Column-oriented database*. Data is stored in expandable columns.
- *Document-based store*. Data is organized in documents with any number of fields.
- *Graph database*. Data is stored in a graph with nodes and relationships between nodes. The values are stored as key value pairs under a node or a relationship.

Normally, NoSQL databases do not fulfill the ACID theorem because they lack full consistency (Leavitt, 2010). However, in NoSQL databases transactions are made usually by using the BASE (Basically Available, Soft state, Eventually consistent) model (Pokorny, 2013; Pritchett, 2008). With BASE the data availability is prioritized over the consistency of the CAP theorem (Pokorny, 2013). In a NoSQL database, complex queries can be more difficult to make as the system is not built for that (Leavitt, 2010).

A NoSQL database should be chosen over a relational database to support more users and have a better performance (J. Han et al., 2011), which are the objectives of the case. On the contrary, a survey by Li & Manoharan stated the performance of a NoSQL database is not necessarily better than a relational database (2013). Different results can be obtained depending on the database product and the case. Therefore, decisions for a database model or product should be made on the requirements of a case.

2.4 Related work

Different studies have been done on comparing different cloud solutions. Höfer & Karagiannis proposed a tree-based structured taxonomy for capturing characteristics of single cloud computing services for quick comparisons between them. They restricted themselves to include only characteristics with clearly distinguishable options (2011). The used characteristics are qualitative metrics as the category of service model, license and payment types, formal agreements (SLA), security measures, standardization efforts, openness of clouds, supported software operating system, tools, services, and programming languages. This model is sufficient for quick comparisons of single cloud computing services but misses several features that do not have clearly distinguishable options. Furthermore, the model looks at single services only and not at a complete architectural solution.

Rimal, Choi, & Lumb compared different cloud computing services provided by cloud vendors in a table regarding qualitative metrics as fault tolerance (availability), security, load balancing and interoperability (2009). The table can quickly show differences between offered cloud computing products.

Li, Yang, Kandula, & Zhang introduced quantitative metrics for comparing cloud computing offers of different public cloud vendors, which are the scaling latency, benchmark finishing time, and cost per benchmark (2010). Scaling latency is the time it takes to turn on or off a computing instance responding to a load.

The focus in these studies is comparing single products of different cloud providers, which can make them quickly outdated as the products are constantly changing. In this study the focus of the comparison are the different architectural designs of cloud computing in general and a comparison of two concrete designed and implemented architectures assessed on the objectives of the case.

3 Application architecture

A typical application in the problem domain consists of user clients on a device or in a browser and the backend application logic in the cloud. A user client is a view of the data retrieved from a database or storage in a cloud and an interface for a user to initiate action requests to the cloud backend. The requests can be sent for example via REST (Representative State Transfer) calls, where the client sends a request and receives a response over HTTP in a standardized format (Christensen, 2009). A server in the cloud processes a request and afterwards it responses the result to the client. An application can have user-related data to which only users have access themselves.

The payment processing is the main functionality in the case of mobile payment application. The payment process is a transaction of different steps; if one action fails the whole payment process fails. For the purposes of this study, the payment processing will be analyzed in the following simplified form. An authorized user is initiating a payment to a cloud endpoint. The endpoint transfers the payment request to a processing logic part. This logic part verifies the payment, makes a payment request to an external banking service, and then stores the payment in a database. Additionally, the application handles several other simple requests for this study as creating, updating, and deleting payment credentials and the retrieving of all made payments of a user. The payment processing and these other functionalities are implemented as cloud backend functionalities, which a user client of the application can initiate. The upcoming case study of the mobile payment application with a main transactional action and the view of data can be transferred to many other similar applications with many users.

4 Architectural approach

There is no cloud architecture standard or single architectural method for designing a cloud backend, but all have the common goal of scalability, availability, and high reliability (Rimal, Jukan, Katsaros, & Goeleven, 2011). Although these are the objectives of the mobile payment case, it is difficult for enterprises to find the correct cloud architectural approach to their specified requirements and constraints (Rimal et al., 2011). Furthermore, nowadays there is a wide and growing variety of different solutions and different public cloud providers, which makes it difficult to decide between them. In this chapter, different common architectural designs are presented and discussed. These architectures can be built on almost every big public cloud provider like Amazon, Google, Microsoft, or IBM.

4.1 Tier-based architecture

The tier-based architecture, also known as layer-based architecture, is one of the most common architecture approaches of software and service development (Urgaonkar, Pacifici, Shenoy, Spreitzer, & Tantawi, 2005), where different parts of the application architecture are divided into tiers or layers to separate them from each other. A standard 3-tier application architecture is divided into a presentation layer, domain layer, and data source layer (Brown et al., 2003; Fowler, 2002).

- *Presentation layer*. Information is displayed to the user and the user can interact with the application by making requests to the domain layer through the presentation layer.

- *Domain layer*. The application logic handles user requests and makes calls to the data source layer. In a cloud architecture, the application logic happens mostly on virtual machines provided by IaaS.

- *Data source layer*. A connection to other systems, which are most commonly a database or a storage for read and write operations.

The logic tier of a 3-tier application in a cloud runs typically on virtual machines (Vaquero, Rodero-Merino, & Buyya, 2011). Traditionally, cloud operators offer isolated virtual machines for computing to have a better server utilization and energy efficiency (Kusic et al., 2009). A controller, which is aware of the loads in a tier, scales the different virtual machines (Kusic et al., 2009). A tier scales horizontally according to the workload of the tier (R. Han, Ghanem, Guo, Guo, & Osmond, 2014). Horizontally scaling means the addition or removal of server instance replicates within a tier (Vaquero et al., 2011). For scaling within a tier, a virtual machine instance takes several minutes to turn on or off (Kusic et al., 2009).

For a 3-tier cloud architecture, the concept of a dispatcher or load balancer can be used to provide a better performance by distributing the load. A load balancer is in front of a tier and distributes requests to different instances of this tier (Urgaonkar et al., 2005). The goal of a load balancer is to improve the performance by dividing the load on different resources to achieve the best resource utilization (Khiyaita, El Bakkali, Zbakh, & El Kettani, 2012).

The cloud architecture of a 3-tier application is depicted in Figure 3. Clients make requests directly or via a REST endpoint to a load balancer, which distributes the load to different virtual machines of the logic tier. Depending on the load, additional VMs can be instantiated to handle the load. In the logic tier, the requests are processed on the VMs. During the processing, the logic tier can interact with the data tier for read or write operations on the database.

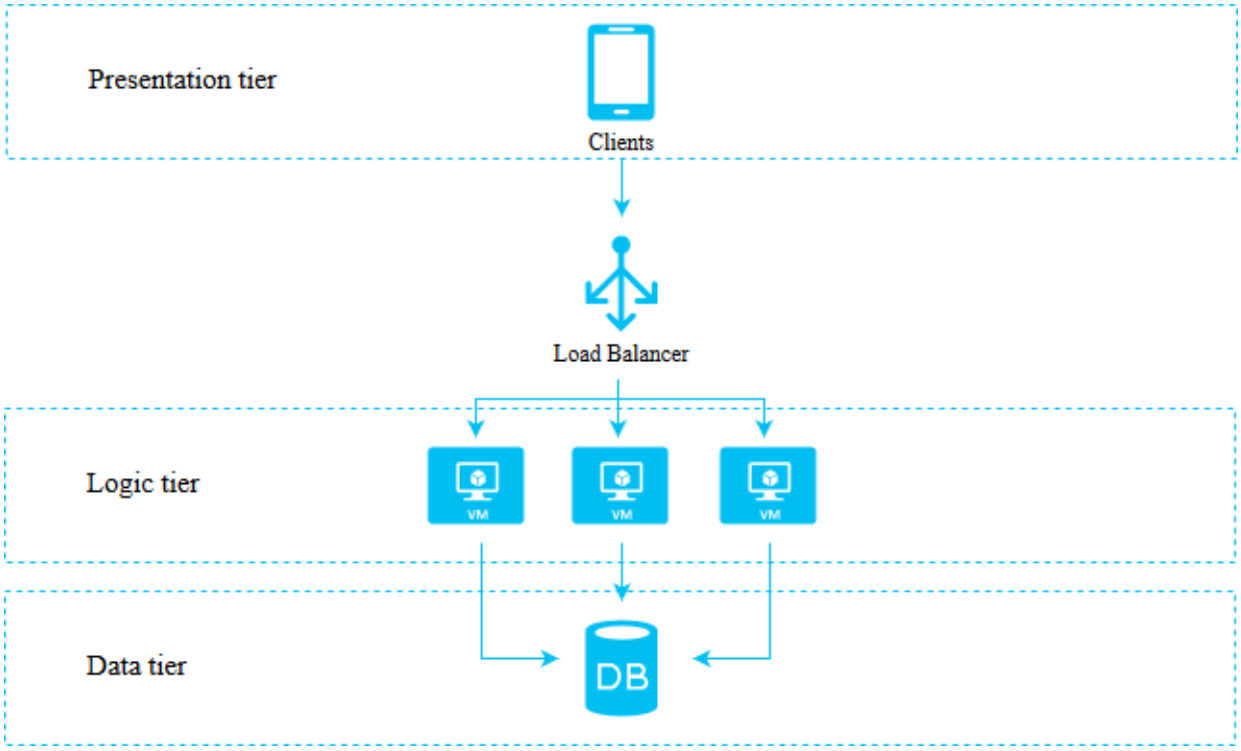

Figure 3. 3-tier cloud architecture

A database in a 3-tier application is traditionally a relational database. The database can be horizontally scaled on demand and is usually built as a replicated cluster with an additional load balancer in front.

The proposed 3-tier architecture can be distributed over several cloud providers in a multi cloud architecture to increase the availability of the system (Grozev & Buyya, 2013). Furthermore, the system can be expanded by adding additional tiers, which leads to the general term of an n-tier architecture for such a system.

Another common model is the 2-tier architecture, which is just divided into a presentation tier and a data tier. Clients are directly connected to the database tier in a 2-tier architecture. Modern examples for a 2-tier architecture are applications for mobile devices and Internet of Things (IoT) devices, which do not have a need for a separated logic tier. (Rahimi, Venkatasubramanian, Mehrotra, & Vasilakos, 2012)

In the case of the mobile payment application, clients making payment requests to the load balancer, which is distributing them to the virtual machines of the logic tier by an algorithm.

If all VMs are under heavy load, a new VM will be initiated by a controller and upcoming requests will be balanced out over the VMs. Each VM will have several clients connected to it and process their payment requests by verifying them, making a request to an external banking service, and storing them after completion to the database. Additionally, all other kinds of possible application requests from the client are handled on the same VM.

4.1.1 Advantages

In a 3-tier architecture the presentation tier, the logic tier and the data tier are strictly divided. The communication within different functionalities of a tier is easy to make and fast, as the complete logic is located at each instance. The client is directly connected to an instance of the logic part, which handles all the requests of the client in a fast way. A payment request is handled on one virtual machine and the client gets directly a response after the payment is successfully processed. The different tiers are separated from each other to make the system more secure. Furthermore, each tier can be developed and tested independently (Brown et al., 2003).

The virtual machines of the logic tier can be configured to the requirements of the application. The developers of the application are not restricted to the platforms or the software offered by the cloud vendor and can design their own infrastructure to their requirements (Baldini et al., 2017). Furthermore, the developers can install updates and patches to the needs of an application.

An instance of a virtual machine can easily be transferred between servers. The danger of having an application that works only on one cloud provider, a so-called vendor lock-in, is minimized as a virtual machine instance can be easily transferred to another public cloud or even to a private cloud. Furthermore, the application can be distributed to multiple clouds, which protects the application from a cloud outage and thus increases the availability.

In a tier-based architecture, a relational database cluster is usually used, which provides high data consistency. Hence, a user can rely on that shown data is always up-to-date. In the case of the mobile payment application, this could for example be that a made payment is directly shown in the payments list of a user.

4.1.2 Disadvantages

The instances of virtual machines are completely scaled horizontally. Hence, some functionalities of the logic tier are so unnecessarily scaled. A high amount of payments would scale the whole application instance in the logic tier. Furthermore, the up and down scaling of virtual machines is slow compared to containers (Joy, 2015), which is problematic for a payment application where the number of users is rapidly changing. VMs need several minutes to turn themselves on (Kusic et al., 2009). Thus, the number of virtual machines must be always higher than the actual demand to handle each payment and to be prepared for rapid changes in the number of users. Hence, the resources of the VMs are not efficiently used by provisioning a higher number of VMs than needed.

The development within a tier happens on the same code base, which makes team collaboration more difficult than developing a more distributed system where functionalities are more separated (Namiot & Sneps-Sneppe, 2014). After a code change in a tier, the whole tier instance must be redeployed to all VMs, which can be difficult and risky for huge changes in the logic (Newman, 2015).

Each different request of an application that is running on the virtual machine is blocking a process during the request processing. Hence, the process cannot be used for other requests. This might be a bottleneck if too many requests are sent to a single virtual machine. Usually, a load balancer does not know the different loads of the different virtual machines and is only distributing the requests according to an algorithm. If requests on a VM are not fast enough processed, requests could accumulate on a VM, which would result in a slow performance. Furthermore, the load balancer is a single point of failure, if it fails requests are not distributed to the virtual machines. This applies also to the other architectural solutions with a load balancer.

If parts of virtual machines have an outage or the number of VMs is not scaled up correctly to the demand, the reliability of the application is in question, as the remaining number of virtual machines might not be able to handle all requests in the same way. The upscaling to overcome this lack of virtual machines takes some time where requests must wait.

4.2 Message queue architecture

A message queue is the central element of a message queue architecture. The message queue organizes and structures the communication between clients and computing instances. The usage of a message queue is a traditional cloud computing architecture approach (Gunarathne, Wu, Choi, Bae, & Qiu, 2011; Malawski, 2016; Satzger, Hummer, Inzinger, Leitner, & Dustdar, 2013).

A message queue can be called by 2 commands, which are adding a message to the end of the queue or removing a message from the beginning (Wilder, 2012). Moreover, a sender is enqueuing a message to a message queue and a receiver is dequeuing and processing a message from the message queue (Wilder, 2012). A message queue can be described as a FIFO-System (First-In/First-Out) as messages are processed in order of their appearance in the queue (Homer, Sharp, Brader, Narumoto, & Swanson, 2014). In this architecture, the queue can be called “pull-queue”, because a receiver takes a message from the queue (Keahey, Armstrong, Bresnahan, LaBissoniere, & Riteau, 2012). Another variant of a queue could be a “push-queue”, where the queue transmits a message to a receiver (Keahey et al., 2012).

A sender and a receiver of a message are loosely coupled, because there is no direct connection between them; thus, there is no need for them to work at the same pace or to wait for each other (Wilder, 2012). A receiver can be a stateless worker, which has no direct information from a sender. Hence, the receiver knows only about the sender and possible task parameters from what is included in the message.

Worker instances in this architecture must be independently able to process a message (Keahey et al., 2012). In case of a failure of a worker instance during processing a message, another worker instance should be able to overtake the message (Keahey et al., 2012; W. Lu, Jackson, & Barga, 2010). To achieve this possibility, workers are not directly removing a message from a queue; instead, they set the message in a process state and remove the message after completing the task (Gunarathne et al., 2011).

In a message queue-based system as in Figure 4, clients send their requests to a web endpoint. The requests can be sent for example via REST. The endpoint transfers the request to

a message and puts it at the end of the message queue. Each idle worker takes a message from the queue in order of appearance. If there is no idle worker for taking a message from the queue, more worker instances are created to handle the demand. Likewise, if there are too many idle workers and no messages in the queue, some instances can be deactivated. A worker processes one message at a time and, if necessary, connects to the database for a read or write operation. After that, the worker can notify the client via the web endpoint about the finished request.

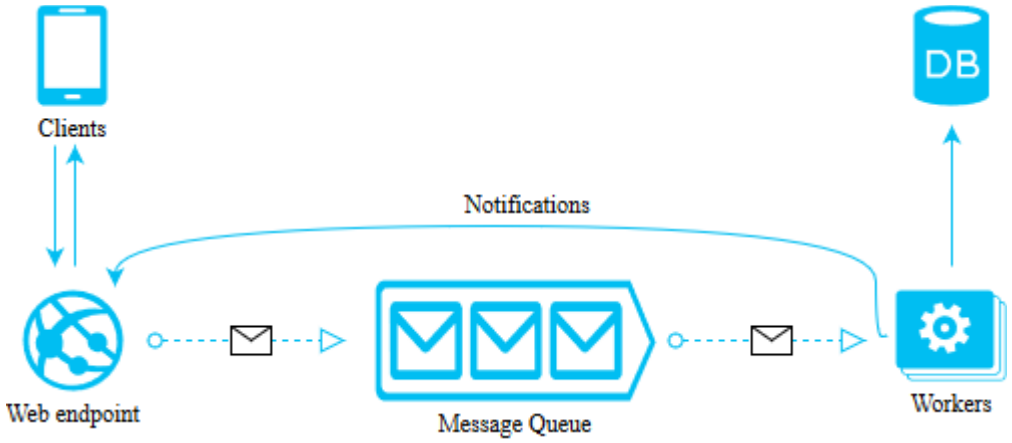

Figure 4. Message queue architecture

For example, a similar cloud architecture with a message queue is used for processing big amounts of healthcare data (He, Fan, & Li, 2013). In such an architecture, workers are usually IaaS or PaaS computing instances. In a message queue architecture, different kinds of workers could be assigned only for certain tasks, so that they would take a message from the queue only if the message correlates to their task.

In the case of the mobile payment application, clients send payment requests to the endpoint, which transfers them as a message to the queue. Then the payments are processed by worker instances in order of appearance. The endpoint, the worker instances, and the database scale according to the number of payments for the payment processing.

4.2.1 Advantages

In a message queue architecture, the workers and the message queue can be configured to the requirements of an application. For example, the message queue could be configured as a priority queue to prioritize different kinds of messages (Homer et al., 2014). For example, in the payment application payment requests could have a higher priority than other functionalities to increase the performance of the payments.

The workers in the message queue architecture could be designed to be responsible for just a certain task and so worker instances are instantiated and deactivated on demand of the certain task. Furthermore, workers of different tasks could so be tested and deployed independently.

In a message queue architecture, there is no need of an extra load balancer, because the load gets naturally distributed with a message queue over worker instances (Gunarathne et al., 2011). Furthermore, a message queue architecture is better protected in contrast to a load balancer against a failure caused by a workload burst, because a message queue provides a buffer by decoupling the web endpoint and the workers (Homer et al., 2014; W. Lu et al., 2010).

A worker and a client are loosely coupled in a message queue architecture and thus they are working at a different pace. Hence, a client does not need to wait for a response from the worker which might take some time (Wilder, 2012).

4.2.2 Disadvantages

In a message queue architecture, worker instances are scaled on demand of an application. If there are no idle workers for a certain task, a new worker is instantiated. Likewise, if there are too many idle workers, worker instances can be deactivated. The needed time for activating and deactivating a worker instance is high and leads to an overprovision of workers and therefore to a wastage of resources.

This architecture type can have a lower reliability as a peak of many messages can cause a high processing time of a request if the workers are not taking the messages from the queue

fast enough. On the other hand, if there are not so many messages, the response time could be faster for a request as the message gets directly taken by an idle worker instance.

In the case of the payment application, the message queue must be reliably configured so a payment request message is only once processed and is not enqueued by several workers. After such a failure of a payment being processed multiple times, a customer might not use the application again.

In a message queue architecture, workers are designed to handle resource intensive tasks or long-running workflows (Wasson, 2017). Hence, in some cases a simple operation could be faster processed without using a message queue and a worker.

4.3 Microservice architecture

Fowler & Lewis define microservices as a development approach to encapsulate a single application into small services, which are functioning on their own (2014). A microservice is a lightweight independent service with a single responsibility and it runs on a single process. A microservice architecture can be described as a specific and better implemented approach of Service Oriented Architecture (SOA) (Newman, 2015).

The counterpart to a microservice architecture is a monolithic architecture where the whole application is a single unit (Fowler & Lewis, 2014). In a monolithic application, a small change results into a redeployment of the whole application (Newman, 2015). The scaling of the whole monolith needs more resources compared to scaling microservices on demand (Fowler & Lewis, 2014; Newman, 2015). The differences between the scaling mechanism is shown in Figure 5. A monolithic application scales completely over several nodes. On the contrary, microservices scale just themselves on the demand of a certain microservice.

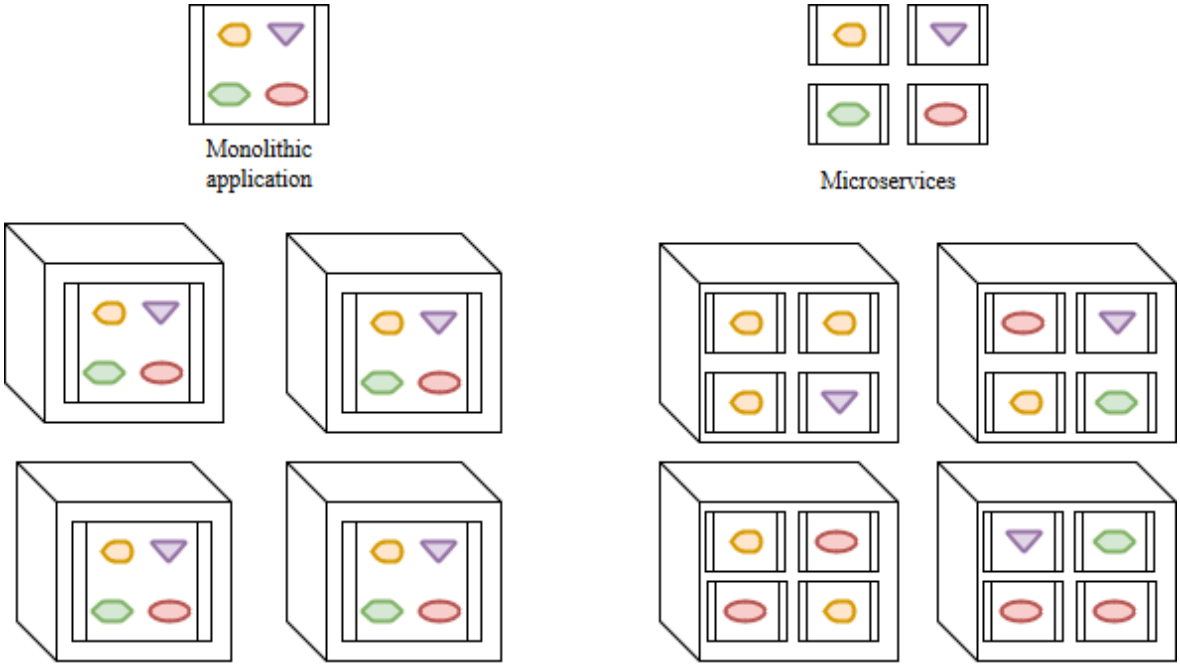

Figure 5. Scaling comparison between monolithic application and micro-services (Adapted from Fowler & Lewis, 2014)

Microservices can be understood as single components rather than libraries (Namiot & Sneps-Sneppe, 2014). Typically, the microservice approach uses the container technology as computing instances (Stubbs, Moreira, & Dooley, 2015). Each microservice is deployed to a single container, which can be deployed to a cloud environment and runs independently and isolated on PaaS (Joy, 2015; Newman, 2015). Furthermore, microservices do not have to share the same programming language; instead development decisions can be made case by case and to the preferences of the developers (Thönes, 2015).

In a microservice architecture, a service registry is needed to keep track of the addresses of different microservice instances, which are instantiated and terminated on different server nodes. A microservice instance registers and deregisters itself to the service registry accordingly to its status. Server-side service discovery is the process, in which a gateway server or load balancer in front of a microservice retrieves the knowledge from the service registry where different microservice instances are located (Balalaie, Heydarnoori, & Jamshidi, 2015). In contrast, client-side service discovery means that a client or a microservice

discovers the address of a microservice from the service registry itself and makes a direct request to the microservice.

Typically, a gateway endpoint receives the client requests and distributes them with the help of a service registry to different microservice nodes. The communication from a gateway or the inter service communication can happen over a lightweight HTTP method (Fowler & Lewis, 2014). In a microservice architecture, the data management can be decentralized over the microservices. This means, each microservice can have its own data persistence (Fowler & Lewis, 2014).

A microservice architecture is presented in Figure 6, clients send their requests to a gateway endpoint, which distributes them to the correct microservice. The address of a microservice instance is obtained from the service registry with server-side service discovery. A microservice processes its requests and stores data to its database if applicable. Then, the microservice can notify the client or calls another microservice for a following task for the request.

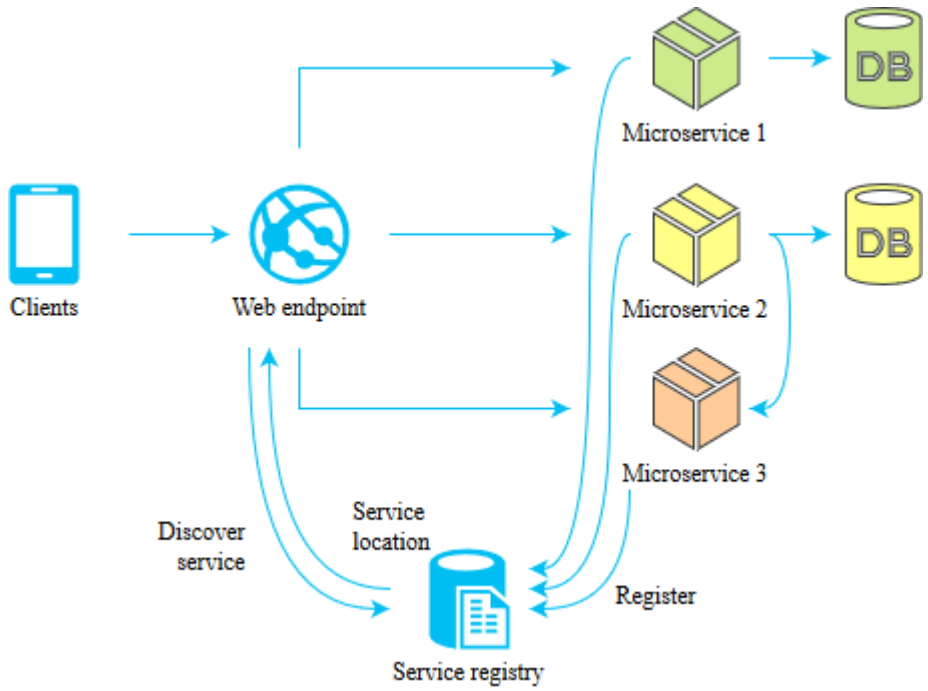

Figure 6. Microservice architecture

In the case of the mobile payment application, different microservices could be CRUD operations on payment credentials, processing of a payment, request to an external banking service, and the storage of a payment.

4.3.1 Advantages

A microservice scales according to the demand of a certain functionality. Furthermore, containers that are used in microservices have a better scaling latency than virtual machines (Joy, 2015). In this way, resources are used more efficiently, and the architecture can better support the rapidly changing user amount in the case of the payment application.

A microservice architecture makes collaborative working and testing of single functionalities easier as each microservice can be handled independently (Joy, 2015; Namiot & Sneps-Sneppe, 2014). Each microservice could be programmed in another programming language according to the preferences of the developing team or the requirements of a microservice (Thönes, 2015). Furthermore, new additional functionalities can easily be added to the architecture by creating a new microservice. In addition, a new microservice can be independently tested and deployed to the cloud if it does not depend on another microservice.

In a microservice architecture, each microservice can have its own encapsulated database. For instance, small NoSQL datastores can be created, to which only certain microservices have access. In the case, different database instances for payments and payment credentials can be created for the different microservices. In this way, databases are more secured and better organized to scale correctly to the demand of a certain request type.

4.3.2 Disadvantages

For a developer it is difficult and might be not possible in every case to divide an application system into smaller microservices (Namiot & Sneps-Sneppe, 2014). In addition, microservices could vary extremely in their sizes, which would omit the benefits of dividing the system into different microservices. For the case of the mobile payment application this is not a problem because the application logic is manageable to divide. Furthermore, it is

difficult for a developer to test the whole system of microservices as it is a distributed system, where different services can have influence on each other (Namiot & Sneps-Sneppe, 2014).

In a microservice architecture, the communication from the gateway to a microservice and the inter service communication must be planned and configured (Namiot & Sneps-Sneppe, 2014), which is an additional workload in the networking layer compared to the other solutions (Thönes, 2015).

If microservices are cascaded in a process, the communication between the microservices happens over a service discovery process, which takes more time than a direct connection or having the process in one microservice. However, a microservice with several functionalities would be against the design pattern of making small microservices with a single responsibility.

4.4 Serverless architecture

A Serverless architecture in the cloud is a relatively new approach. Serverless does not mean that there are no servers, the term defines itself that there is no need for the cloud consumer to create or maintain servers, which is completely and automatically done by the cloud provider (Baldini et al., 2017). Serverless technologies are offered as platforms by cloud vendors between the traditional service models of SaaS and PaaS (Fox, Ishakian, Muthusamy, & Slominski, 2017). Hence, the Serverless approach is located on a higher service model level than the microservices approach, which is working completely on PaaS. In a Serverless approach there is no need for the cloud consumer to monitor and manage different microservice instances and to setup the communication between them.

The Serverless approach can be described with the term of Function as a Service (FaaS) as part of the widely used “as a Service” terminology (aaS) (Duan et al., 2015). Thus, so called functions can be triggered by different multi-protocol events and are executed in an asynchronous or synchronous way (Spillner, Mateos, & Monge, 2017). The different triggers for a function can be for example to write operations to a database, a REST call, or to write operations to a storage. In addition, a function is mostly stateless, which can retrieve data

during runtime or is called with parameters. There is a discussion ongoing if a function could be stateful in future (Baldini et al., 2017; Fox et al., 2017).

A common example of a Serverless function, which has been named the “Hello World” of Serverless computing (Baldini et al., 2017) is displayed in Figure 7. An image gets uploaded to an image store, this triggers the Serverless function, which is automatically generating a thumbnail of this image, and stores the thumbnail in the storage.

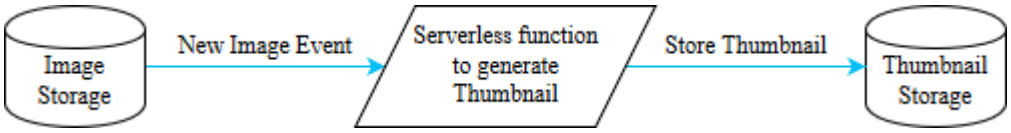

Figure 7. Serverless function thumbnail generation (Adapted from Baldini et al., 2017)

An instance of a function is running and thus scaling on demand of the function (Fox et al., 2017). When an instance is provisioned the first time, it will be served via a cold start, which can cause a delay in the execution time. When the function is regularly used, the function is ready to run and triggers without delay. Generally, a function has a limited short runtime of 5 to 15 minutes. Therefore, a longer task must be divided into several functions (Baldini et al., 2017).

A Serverless architecture is depicted in Figure 8, where clients make a request to an endpoint. The request can cause a REST function trigger, which activates a function to run. The function can interact with a database during runtime and can so trigger another function.

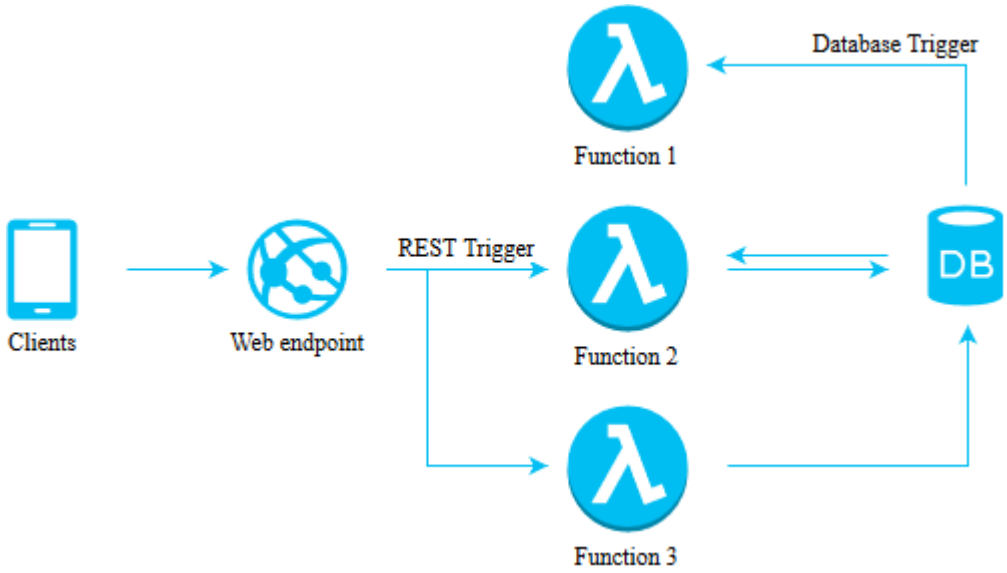

Figure 8. Serverless architecture

For the case, the application logic can be split in a similar way as in the microservice approach. The functions in the serverless architecture have additional possibility to be triggered by different events. For example, a payment could be written to the database, which triggers the payment processing function to run in the cloud.

The underlying technology of a Serverless approach is presented by McGrath & Brenner with a prototype that is utilizing two message queues and the functions are running in containers (2017). Hence, the Serverless technology is a further development of other cloud architectures, which makes the setup easier for the cloud consumer. In other studies, different solutions of FaaS have been tested to each other. For example, a performance test has been made between different FaaS in different scientific computing domains (Spillner et al., 2017). Furthermore, a concurrency test has been made on different Serverless computing implementations from different public cloud vendors and a self-created Serverless prototype (McGrath & Brenner, 2017).

4.4.1 Advantages

The scaling of a Serverless environment happens automatically by the cloud provider without interaction or configuration from the cloud consumer. Hence, up- and downscaling is

fast, because the cloud provider optimizes the system and a function is a small computing instance. Furthermore, resources are not wasted and a cloud consumer pays only for the execution time of the function and per invocation (Baldini et al., 2017). Idle times of a function are usually not charged by the cloud provider, which makes the approach attractive for companies with an unpredictable number of users or, in many cases, without any active user.

The public cloud provider handles the configuration and maintenance of servers in a Serverless environment. Hence, a cloud consumer can concentrate himself on the code production of an application (Baldini et al., 2017). There is no need to configure the network communication between functions like in the microservice architecture. Furthermore, new functionalities can be easily created by the developer and added as a new function to the application without changing other functions.

4.4.2 Disadvantages

In a Serverless architecture, a function can have a slow performance if it happens to be a cold start of the function (Baldini et al., 2017). This could be a problem for a performance-oriented function, which is not triggered frequently. This problem can be overcome by keeping a function instance running with dummy requests. Such dummy requests are sent regularly to a function, which recognizes them as a dummy request and discards them. However, the provisioning of a function instance causes the usage of extra resources.

At some point the cloud consumer might face the problem of a vendor lock-in for an application created in a Serverless environment (Baldini et al., 2017). That means that the generated code only works with the chosen public cloud provider and it is not possible to change the cloud provider without rewriting the code. In the other solutions, container and virtual machines can be more easily transferred between cloud providers. Furthermore, the offered Serverless environment by a cloud vendor might not be sufficient enough to the requirements of a cloud consumer, because the environment cannot be configured or changed according to the needs of the cloud consumer.

Currently, FaaS do not support longer tasks, because a single function has a runtime limit. Hence, longer tasks must be split over several different functions (Baldini et al., 2017). For

the case of the payment application that is not a problem, because there is not any long-running task yet.

5 Assessment of the different cloud architectures

The review of different architectural approaches shows that each approach has their pros and cons, but they have also similarities in their architectural style of organizing the application into different parts. Furthermore, the different architecture designs have the same goals of fulfilling the objectives of the case. In the order of appearance of the different approaches, the progress of the development of architectures in cloud computing can be seen. The progress goes from bigger computing units and more configuration possibilities of servers by the cloud consumer to more smaller computing units and no configuration at all. In the following assessment, a decision for implementing a solution is based on the requirements of the case with the assessment criteria of availability, scalability, reliability, and needed resources.

The tier-based architecture has a high availability and has been proven to be a reliable concept over years. In contrast, the scalability of a tier-based architecture is the worst compared to the other architectures, because the biggest computing instances in form of virtual machines are scaled on demand in a tier. Furthermore, virtual machines have a high scaling latency, which means they need several minutes for up- and downscaling an instance, which might be too slow for a rapidly changing number of users. Hence, the needed resources for a tier-based architecture are higher, because the provisioned resources must be higher than the actual load to be able to adjust to rapid user changes. However, the setup of a tier-based architecture is easily done and is a standard process in software development.

The message queue architecture is as well a proven and reliable concept in cloud computing and profits from organizing the communication between clients and worker instances in a structured asynchronous way. Additionally, a queue is less likely to fail than a load balancer of other architectures on a bursting workload, because the queue buffers naturally requests into messages and the workers process the messages successively. For that, worker instances are scaled on the throughput of messages in the queue. However, the scalability could be better if the architecture would be built more like the microservice approach with several message queues and own pools of worker instances for certain responsibilities to scale different parts of the architecture accordingly to a certain functionality. Otherwise, this

architecture uses more resources for scaling a worker. Furthermore, the setup and configuration of a message queue and worker subscription is an additional work load for a developer.

The microservice architecture structures an application into lightweight services that should work and run independently from each other. Hence, team collaborations and testing of single functionalities in a microservice architecture are easier to do than in a more monolithic architecture. The scaling of microservices is caused by the demand of a certain microservice. In this way, resources are not scaled unnecessarily. Additionally, in a microservice architecture a datastore can correspond to a single microservice to have a better performance and security. On the other hand, it is more difficult to build an application into different microservices with single responsibilities, and therefore more work time is needed. Furthermore, more resources are needed, because a service discovery method and service registry must be planned and configured for the communication between and to different microservice instances in this architecture. The performance in this architecture can be lower than a more monolithic architecture for transactions, which use different microservices during the process instead of a single machine.

The Serverless architecture makes it easier for a cloud consumer to concentrate on the application logic, because the cloud provider handles the configuration and the maintenance of servers. Therefore, the needed resources for the setup and the maintenance are low. The scalability is as good as in the microservice architecture by scaling just the function on demand of the load on this function. Furthermore, the scaling latency is low, because the cloud provider optimizes the up- and downscaling of function instances. In contrast, a Serverless architecture can still have certain launch difficulties that are not solved yet and thus the reliability is lower than in the other architectures. For example, FaaS has a low performance if a function has a cold start, because the function is not triggered regularly.

The architecture of an application can be built on multiple clouds of different cloud vendors to have a better availability overall and so to overcome a single point of failure of a cloud outage (Armbrust et al., 2010). Furthermore, a vendor lock-in can be avoided by building the application as a multi-cloud system. A multi-cloud system can most easily be achieved with a tier-based architecture. In contrast, serving the application in different clouds would

result in higher costs and in more maintenance work. The availability depends also on reduction of single point of failures. Hence, load balancers and web endpoints must be able to handle a high number of client requests and should not be prone to failures.

The assessment of the different architectural solutions is summarized in Table 1 with a grading in the different criteria. The tier-based architecture has the highest availability amongst the solutions, because it can be easily deployed to different clouds. The best scalable solutions are the microservice architecture and the Serverless architecture, because they are scaled to certain functionalities and have the lowest scaling latency. The best reliability is assured in the tier-based architecture and message queue architecture. The needed resources are the lowest in the Serverless architecture, because the consumer can directly use the solution without setting up and configuring the environment. Furthermore, a Serverless architecture is only charged for the running time of computing units and not for idle times.

	Availability	Scalability	Reliability	Needed resources
Tier-based architecture	High	Low	High	High/Low
Message queue architecture	High/Low	High/Low	High	High/Low
Microservice architecture	High/Low	High	High/Low	High/Low
Serverless architecture	High/Low	High	Low	Low

Table 1. Assessment of the different cloud architectures

The company of the case has initially chosen a Serverless approach in Google Firebase, which is a good first choice for the case due to the fact that for a company a Serverless

architecture is easy to implement and so is not requiring that many resources. Furthermore, the architecture is provisioned on the demand of the application and has no fixed costs.

In this thesis, the microservice architecture will be implemented alongside the Serverless architecture and compared to it in favor of the other solutions, because the resource utilization in scaling is better in the microservice architecture than in the other two more traditional solutions. Another factor is the organization of the application into small independent parts with a single responsibility, which makes the application organized and easily extendible. Furthermore, the microservice architecture and Serverless architecture have not yet received much attention in the research, despite the fact that they are the current trends of cloud computing. Additionally, the approaches are fitting well to the lightweight mobile payment application case and other applications in the same domain with a rapidly changing number of users.

6 Products Presentation

In this chapter, the different products or services are presented, which are used or available for the following practical implementation section of the Serverless architecture and the microservice architecture. At first, general products are introduced and then the products of Google Firebase and Amazon Web Services are explained as computing services, databases, authentication services, developer tools and communication services.

6.1 General products

The programming code of the practical implementation is mostly written in JavaScript. Additionally, Node.js is used as a JavaScript runtime environment for executing JavaScript server-side code. Alongside, the package manager npm is utilized, which offers a great variety of additional software libraries. Other programming languages could also be used in the implementation if they are applicable or supported by the cloud provider.

6.1.1 Express[1]

Express is a robust and flexible Node.js web application framework created by the Node.js Foundation (Express, 2018). Express is used as a RESTful web service interface in the practical implementation. HTTP requests to an Express web service could be for example GET and POST requests. A GET request is retrieving values. In contrast, a POST request is submitting values for processing them. Express can work with the JSON data format, which is mainly used in the practical implementation to store and exchange data between services. An Express web service is easily implemented and organized.

6.1.2 Docker[2]

Docker is a software containerization platform that builds, secures, and manages applications in containers (Docker, 2015). A Docker container can be run and tested locally by leveraging

[1] Express is available on https://expressjs.com/
[2] Docker is available on https://www.docker.com/

virtual resources of a local machine. Almost every software application can be built into a Docker container image. Hence, a developer or a developer team have the freedom to choose the programming language or software for a certain application or service according to their needs.

6.2 Firebase products[3]

Google Firebase offers an easy to implement software development kit (SDK) for different mobile devices and the web. Firebase can be assigned to the service model of Backend as a Service (BaaS), because clients connect directly to the backend services (Roberts, 2016). All resources of services in Firebase are provisioned on demand. In Firebase, it is not possible to configure or change the underlying infrastructure of different services. The services of Firebase run on resources of Google Cloud Platform, since Google acquired Firebase in 2014. Firebase is currently only available in the Central US region.

6.2.1 Firebase Realtime Database

The Firebase Realtime Database is a NoSQL database document store. The Realtime Database is cloud-based and schema free. In the Realtime Database, data is saved in a JSON tree and data nodes are synchronized to all listening clients in real-time. Clients use the Firebase SDK to access directly the Realtime Database without using an application server in between. The Realtime Database can offer offline capabilities through database persistence on the disk of the client. (Firebase, 2018e)

Clients use a database reference for listening and writing to a certain data node. The Firebase Realtime Database is built for non-complex query operations that are quickly executed. It is advisable to flatten the data structure as there is a limit of nested levels and all children get additionally requested if a parent gets queried. Data can be secured and validated in the Realtime Database with a set of rules on each data node. Furthermore, indexes can be defined for faster querying of a certain data node.

[3] Firebase products are available on https://firebase.google.com/

6.2.2 Firebase Cloud Firestore

Firebase Cloud Firestore has been introduced in October 2017 and is currently in beta state. The Cloud Firestore is a document model NoSQL database and therefore the newest alternative to the Firebase Realtime Database. In the Cloud Firestore, data values are stored with field mappings in documents. Each document can be part of a collection. Collections can be structured hierarchically with subcollections. (Firebase, 2018b)

Unlike the Realtime Database, in Cloud Firestore there are more complex expressive queries possible. For example, shallow queries can request data at the document level without retrieving the collection or subcollections. Furthermore, a Cloud Firestore database instance should support more concurrent connections than a Realtime Database instance and Cloud Firestore is planned to be available in multiple regions after the beta phase.

6.2.3 Cloud Functions for Firebase

Cloud Functions for Firebase is an event-based FaaS. Cloud Functions for Firebase was introduced in 2017 and is currently still in beta phase. Functions are written in the Node.js environment and with the Express server framework in the programming languages JavaScript or TypeScript. (Firebase, 2018c)

An event triggers a single function. Different kinds of events can be a Realtime Database trigger, Cloud Firestore trigger, HTTP trigger, Cloud Storage trigger, or a Cloud Pub/Sub trigger. Firebase manages and scales function instances automatically according to the load. Hence, a cloud consumer can concentrate on the function logic.

6.2.4 Firebase Authentication

Firebase Authentication is a ready to use authentication service. It gives several options for identifying a user as mail with password, different federated identities (Google, Facebook, Twitter), or phone authentication. (Firebase, 2018d)

The Firebase Authentication is easily implemented with the Firebase SDK on clients. The Firebase SDK provides sign-in, sign-up and password change methods. The Firebase Authentication can be used with the Firebase databases to secure personal data.

6.3 Amazon Web Services (AWS) products[4]

AWS provides several different kinds of services for a great number of cases rather than Google Firebase, which is concentrated with its services on the mobile application market. Furthermore, in AWS it is possible to configure infrastructures to the needs of an architecture.

6.3.1 Amazon Elastic Container Service (ECS) and AWS Fargate

Amazon ECS is a container orchestration service for containerized applications built with Docker (AWS, 2018d). Amazon ECS manages and runs containers on an Amazon Elastic Compute cluster. The container instances can be scaled horizontally by adding new container instances to the cluster.

A cloud consumer had to plan and configure an Elastic Compute cluster infrastructure himself before the release of AWS Fargate in December 2017. AWS Fargate gives the possibility to use a cluster in a Serverless automatic way, thus the cluster is configured, and its resources are provisioned by AWS without interaction of the cloud consumer. AWS Fargate is currently only available in the North Virginia Region. (AWS, 2018h)

6.3.2 Amazon Elastic Container Registry (ECR)

Amazon ECR stores, manages, and deploys easily Docker images in a repository (AWS, 2018f). Docker container images are pushed to an Amazon ECR repository and thus Amazon ECS can use them for deployment. The different versions of images are organized in a repository with tags.

[4] AWS products are available on https://aws.amazon.com/

6.3.3 AWS Virtual Private Cloud (VPC)

AWS VPC is a logically isolated section of the AWS cloud, where the cloud consumer can control the network environment (AWS, 2018e). Different AWS services can be put in a VPC, therefore there is no direct access possibility from the internet, which increases the security of a service. Clients can make requests to a service in a AWS VPC with a VPC Link or via a VPC endpoint.

6.3.4 AWS Network Load Balancer

An AWS Network Load Balancer is one of the options of Elastic Load Balancing in AWS besides an Application Load Balancer and a Classic Load Balancer. The communication to and from an AWS Network Load Balancer works on the TCP layer (AWS, 2018j). TCP is a lower layer in the OSI model than HTTP. The Network Load Balancer is not opening or changing the HTTP part of a request. Furthermore, a Network Load Balancer can have only a single target group defined, to which the load is distributed, unlike the other Elastic Load Balancers, which could have several different target groups defined.

6.3.5 Amazon API Gateway and Amazon Cognito

Amazon API Gateway is a web endpoint for clients. Different AWS resources can be provided with the RESTful API. Different method resources are called with path-based HTTP/HTTPS requests. In the API Gateway throttling and caching can be configured (AWS, 2018a). The API Gateway scales itself automatically to the demand. The communication to other AWS services can be achieved with a proxy or without via HTTP, VPC Link or Lambda Integration. The API Gateway can be authorized with the usage of an Amazon Cognito user pool. The Amazon Cognito user pool can be a federated identity or a new generated mail and password user. For token identification a JSON Web Token (JWT) is generated for the user after logging into the system.

6.3.6 Amazon DynamoDB

Amazon DynamoDB is a NoSQL database, which supports key value stores and document-based stores. Additionally, a database instance of a region can be divided into different tables. A DynamoDB table is by default for reading eventually consistent but can be configured to be strongly consistent with higher costs. User data can be secured in DynamoDB with Fine-Grained Access Control by the usage of role policies of the AWS Identity and Access Management. (AWS, 2018c)

Amazon DynamoDB provides certain read and write capacity for a table, which can be adjusted to a certain load with AWS Auto Scaling. A DynamoDB table can be configured to only allow connections from a VPC.

6.3.7 AWS Lambda

AWS Lambda is a Serverless FaaS that runs server backend code without configuring and managing a platform or infrastructure. AWS Lambda is provisioned on the consumed compute time. AWS Lambda functions can be written in Node.js, Python, Java, C#, and Go (AWS, 2018i). AWS Lambda functions can be triggered by several different AWS services for example an Amazon DynamoDB event or a REST call.

6.3.8 AWS Cloud Watch and AWS Auto Scaling

AWS Cloud Watch can monitor different kinds of metrics of different AWS services. A metric could be the memory consumption or CPU usage (AWS, 2018b). Alarms can be configured in Cloud Watch for certain events or threshold crossings. The alarms can notify the cloud consumer or take an automated policy action like AWS Auto Scaling. AWS Auto Scaling scales different AWS services like DynamoDB or ECS with different policies and plans according to the alarms provided by AWS Cloud Watch (AWS, 2018g). AWS Auto Scaling should be configured to adjust the resources correctly to the current demand. However, in most cases resources must be overprovisioned, because it is difficult to predict the load and therefore to react in time with correct amount of resources.

7 Practical Implementation

The case of the mobile payment application is implemented on two different cloud providers with two different backend architecture approaches. The Serverless architecture is implemented in Google Firebase and the microservice architecture is built in Amazon Web Services. The architectures are assessed on the objectives of availability, scalability, reliability, and needed resources. Especially, the ability to process payments is studied within the solutions. Additionally, a cost estimation is given for the implementations on the different cloud providers. Furthermore, drawbacks of the implemented architectures are stated, and possible improvements are presented.

7.1 Serverless architecture in Google Firebase

The company of the case has initially chosen a Serverless architecture in Google Firebase, because Firebase offers a quick development of a first solution with paying on-demand and not having any fixed costs. The Serverless architecture in Firebase is a ready to use environment that does not need much configuration and planning of the architecture. On the contrary, the configuration possibilities are restricted in Google Firebase.

7.1.1 General architecture

The implemented Serverless architecture in Google Firebase is presented in Figure 9. Users authenticate themselves with their preexisting accounts to the Firebase authentication. Clients are directly connected via the Firebase SDK to the Firebase Realtime Database, which handles read and write requests and stores the application state. Read and write events of the Realtime Database can trigger Cloud Functions for Firebase to run. The Cloud Functions for Firebase act as servers in the cloud and run the code that is written in JavaScript with the Node.js runtime environment. Firebase scales the different running services automatically as the Realtime Database or Cloud Functions for Firebase according to the current load.

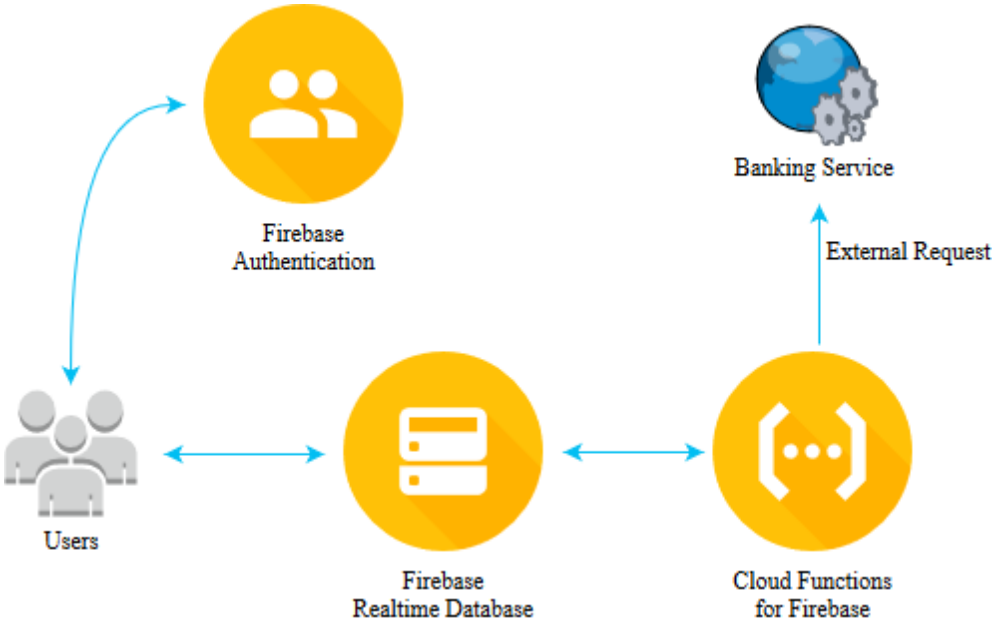

Figure 9. Serverless architecture in Firebase

7.1.2 Payment processing in the Serverless architecture in Firebase

The payment process in the Serverless architecture in Firebase is depicted in a sequence diagram in Figure 10. A user logs in to the Authentication system. Then a user can push a new payment directly to the Realtime Database, which triggers the Payment Process Function with a Create Payment Event. The Payment Process Function reads the payment credentials of the user from the database. The payment credentials are used to authorize a payment. The Payment Process Function makes a Payment Request with the credentials to an External Banking Service. If the payment got successfully processed, the payment in the database gets updated with a success state. The client listens to Realtime Database updates to the payment and can notify the user about the status change of the payment. In this architecture, the focal point for the payment processing is the Realtime Database.

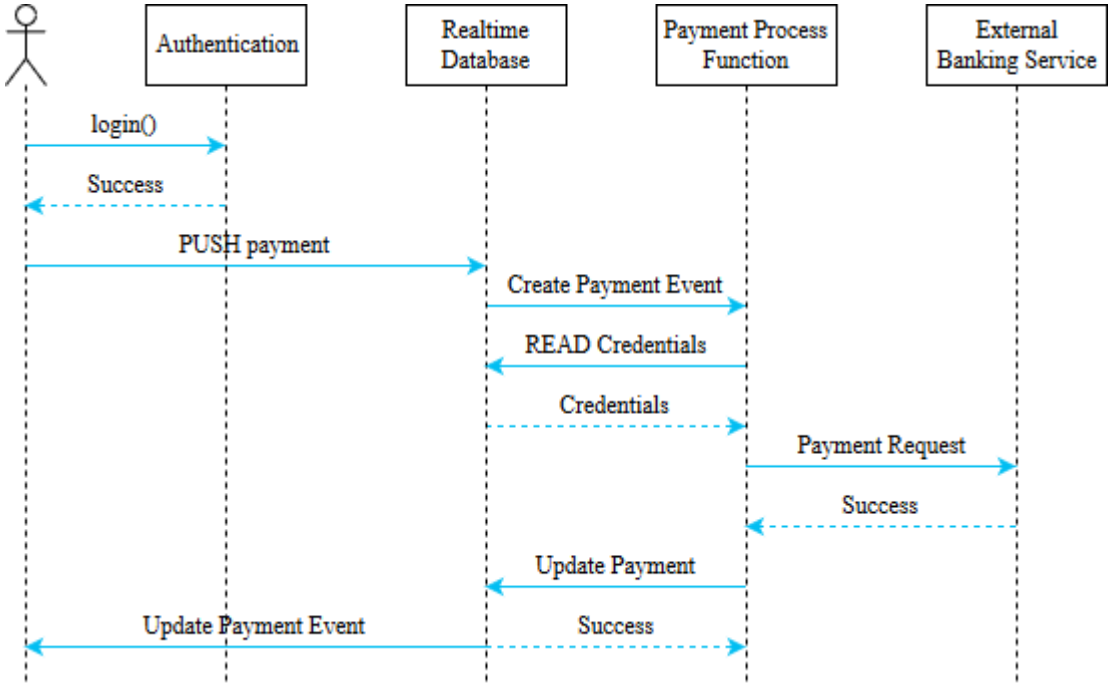

Figure 10. Sequence diagram payment process in Firebase

7.1.3 Realtime Database as a focal point

The Realtime Database is organized as a JSON tree and structured in a flattened way. For example, payments are stored directly under the root node in a payments node. Data validation and access rules can be defined for each data node in a rules file for the Realtime Database. Furthermore, in the rules file the id of a user is set as an additional index besides the primary key. The index can be used for grouping the data values, for example for querying quickly all payments of a user. Additionally, in the rules file authorization rules are set so a user can only write a new payment for himself and can only read his own payments.

The Firebase Realtime Database has a limit of 100,000 concurrent connections and 1,000 write operations per second in a single database instance (Firebase, 2018a). The limit can be increased by sharding the data in several database instances. Sharding the data means splitting the different data nodes in so-called shards in different databases instances. For example, the payments node could be in a separated shared instance of the Realtime Database to increase the limits of concurrent connections and write operations per second. Furthermore,

the clients would need to be configured to connect to the correct database for different database operations.

7.1.4 Assessment

The availability in the Firebase architecture is currently only ensured with a SLA for the Firebase Realtime Database with a monthly uptime percentage of at least 99.95% that corresponds to an allowable downtime of around 22 minutes in a month. A downtime is counted only for continuous five-minute periods. If the SLA is not fulfilled the cloud consumer gets a compensation of up to 30 % of the monthly costs of the Realtime Database in service credits for Firebase (Firebase, 2018f). Cloud Functions for Firebase have no SLA defined yet as the product is in beta state. In summary, the availability in this architecture is in question, because new services in beta state without a SLA are used and the Realtime Database as the focal point of this architecture, which shuts down the application in case of a failure.

The scalability of this architecture has an excellent resource utilization by provisioning automatically the correct amount of resources to the different Firebase services. The scaling of resources is fast but has certain scalability limits for example in concurrent client connections to this architecture. There is a connection limit for an instance in the Firebase Realtime Database of 100,000 connections at the same time and 1,000 write operations per second. Furthermore, for Cloud Functions for Firebase there is a strict limit of 1,000 concurrent invocations per background function and 1,000 invocations per second per function (Google Cloud Platform, 2018). Hence, in this architecture only 1,000 payments can be currently processed in a second, which means there is a maximum of around 2.59 billion payments in a month.

The reliability of this architecture can be seen in its performance, which is mostly acceptable for the payment processing. The performance drops only to an unacceptable level if a Cloud Functions for Firebase encounters a cold start of a function. A cold start for a function occurs if the function is not regularly triggered and thus there is no ready-to-use instance of the function. The instantiation of the first instance of a function will take several seconds, which results in a short delay for the client. The other aspect of the reliability is the data security of

the personal data of a user. In this architecture, data security is ensured through Realtime Database rules that only give the user access to his personal data. However, all clients are connected to the same Realtime Database instance that stores all different kinds of user data. In the case that the database rules do not work correctly or are falsely configured, all personal data could be leaked from the database instance.

The needed resources in this Serverless architecture are low, because the ready-to-use Firebase environment can be directly used with a low work load for the setup and the configuration. Furthermore, the needed time for deploying adjustments and extensions to the application logic is low.

7.1.5 Cost estimation[5]

Table 2 shows a rough estimation of the costs for the designed architecture for 100 million payment requests of the case for a month with the usage of free tiers. A payment is in JSON-format around 200 Bytes. In this calculation, the Realtime Database stores 100 million payments. For each payment, there is one payment read operation and one read operation of the user credentials from the Realtime Database. For the calculation, it is assumed that Cloud Functions for Firebase use the values of a simple function of 128 MB of memory and 200 MHz of CPU per invocation (Google Cloud Platform, 2017). The average running time of the Payment Process Function is 500 milliseconds. An outbound request to the banking service is assumed to be 1 KB. For this architecture, the estimated costs of $297 are relatively small to the high number of 100 million payment requests.

[5] Usage of Firebase Calculator in February 2018 available on https://firebase.google.com/pricing/

Costs for Realtime Database	in $
100 million * 200 Byte ~ 18.6 GB stored	95.00
100 million * 200 Byte * 2 ~ 36.8 GB transferred	40.00
Costs for Cloud Functions	
100 million invocations	39.20
(128/1024) x 0.5s = 0.0586 GB-seconds per invocation * 100 million ~ 5.86 million GB-seconds	14.00
(200/1000) x 0.5s = 0.100 GHz-seconds per invocation * 100 million ~ 10 million GHz-seconds	98.00
1KB request * 100 million ~ 95 GB Egress	10.80
Total estimated costs	**297.00**
Price per payment	**0.00000297**

Table 2. Cost estimation Firebase implementation

7.1.6 Drawbacks and possible improvements

The Firebase Realtime Database has scalability limits in the form of concurrent connections to it. The Realtime Database can be shared in several instances to have a higher limit of concurrent connections overall. Another possibility to solve this problem would be a switch to the Cloud Firestore database as soon as it leaves the beta state and has a higher scalability available. If the case of the mobile payment application would reach the limits of too many users for database, this problem should be solved.

Another limit of the scalability is the limit of 1,000 invocations per single function per second. This could be improved by splitting or cloning the Payment Process Function in several functions, which have different trigger events and are working independently from each other. The configuration of such a system on the client and in the cloud would mean an increased work load, which should be considered if payments are concentrated to a specific time and therefore break the limit of 1,000 payments per second.

In this Serverless architecture, cold starts of functions are a problem after time periods, where a function is not regularly triggered. Cold starts can be overcome with the implementation of dummy requests, which trigger the function regularly and keep at least one function instance always up. This solution would result in a higher resource usage and therefore higher costs for the cloud consumer. Hence, this improvement of the performance can be considered from case to case, where a certain performance is always required.

7.2 Microservice architecture in Amazon Web Services

Amazon Web Services (AWS) is chosen as a public cloud provider for the microservice architecture, because AWS is currently the biggest public cloud provider with a market share of about 40 % (Synergy Research Group, 2017) and gives a great variety of cloud services. A microservice architecture in AWS can be designed in various ways, because the concept of a microservice, being a small computing unit with a single responsibility, can be implemented with different AWS services. For this study, a traditional microservice approach with containers as computing units is selected.

7.2.1 General architecture

The practical implementation of the microservice architecture in AWS is depicted in Figure 11. A user logs in the authentication system Amazon Cognito with his user credentials over the API Gateway. The API Gateway is a REST endpoint and offers in the case log-in methods and the different microservice methods. A user can call the microservices with authorized HTTPS requests. The authorization happens with a JSON Web Token (JWT) in the HTTPS header. The API Gateway validates the token against an Amazon Cognito user pool.

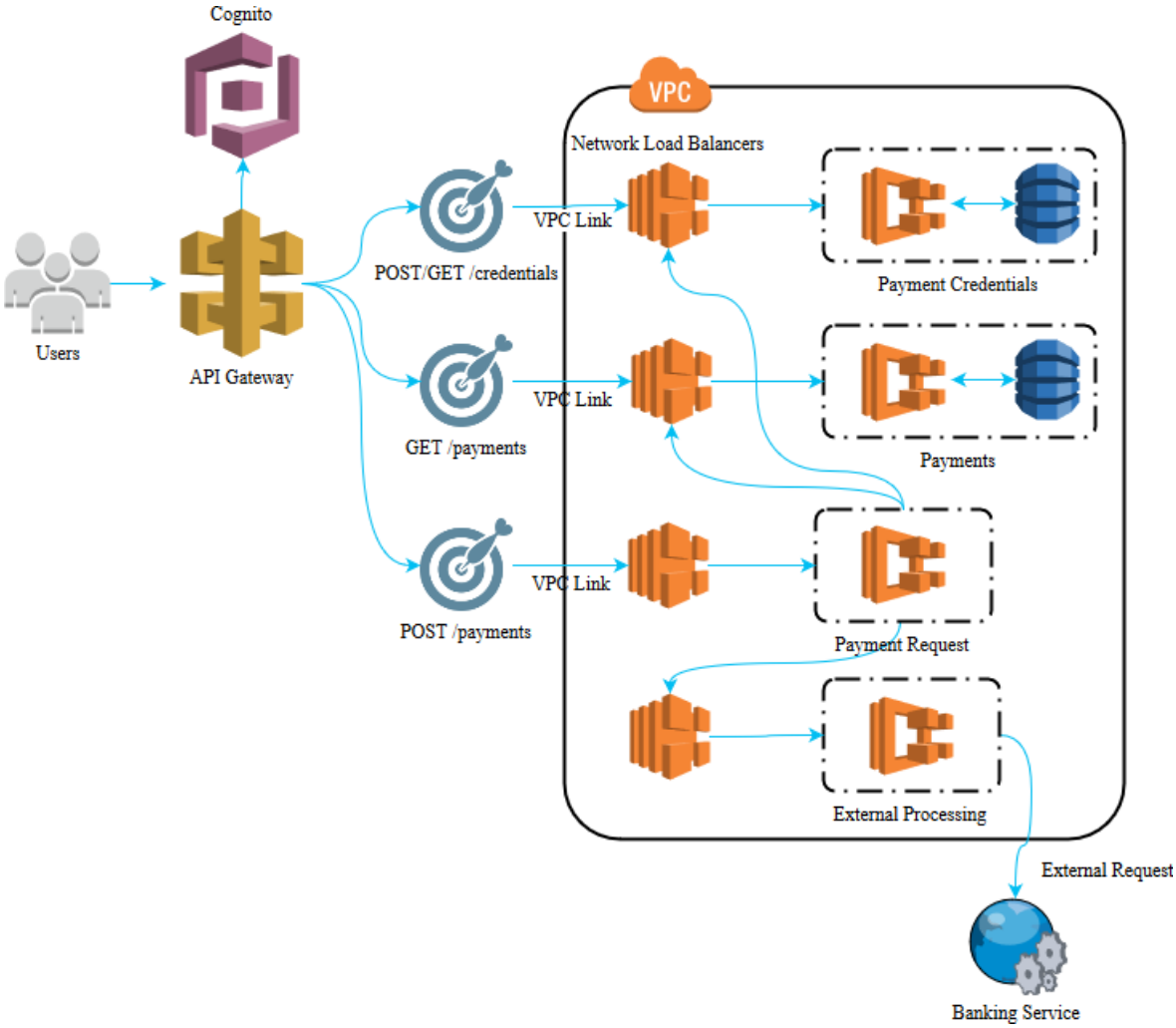

Figure 11. Microservice architecture in AWS

The microservices are situated in a Virtual Private Cloud (VPC) and thus they are not reachable directly through the internet, which increases the security of the implementation by making the attack surface smaller. The API Gateway transfers only authorized user requests to the different microservices over a VPC link proxy to an internal Network Load Balancer. As a result, there is no need for a microservices to validate the authorization again with Amazon Cognito, when a request reaches a microservice.

The internal Network Load Balancer distributes the requests to a target group in AWS. A target group consists of different container instances of a microservice. A request is processed on a container instance of a microservice. During the request processing, a microservice can be stateless or connect to a Dynamo DB table for a read or write operation. In

this architecture, DynamoDB tables are not shared between the microservices. Therefore, a Dynamo DB table scales itself according to the demand of a certain microservice.

7.2.2 Payment processing in the microservice architecture in AWS

The payment process of the payment application case is shown in Figure 12 as a sequence diagram. A user logs in with his Cognito account and initiates a payment with a POST request. The request includes the authorization JWT token in the HTTPS header. The API Gateway validates the token with a Cognito method. After the token validation, the payment request is sent to the Payment Request Service. The Payment Request Service makes a GET request to the Credentials Service to get the user payment credentials for authorizing the payment. After the credentials have been retrieved, a request is sent to an External Processing Service for handling the payment request with an external banking service. If the banking service approves the request, a request is made to the Payment Service to store the payment. Then the Payment Request Service notifies the user that the payment succeeded.

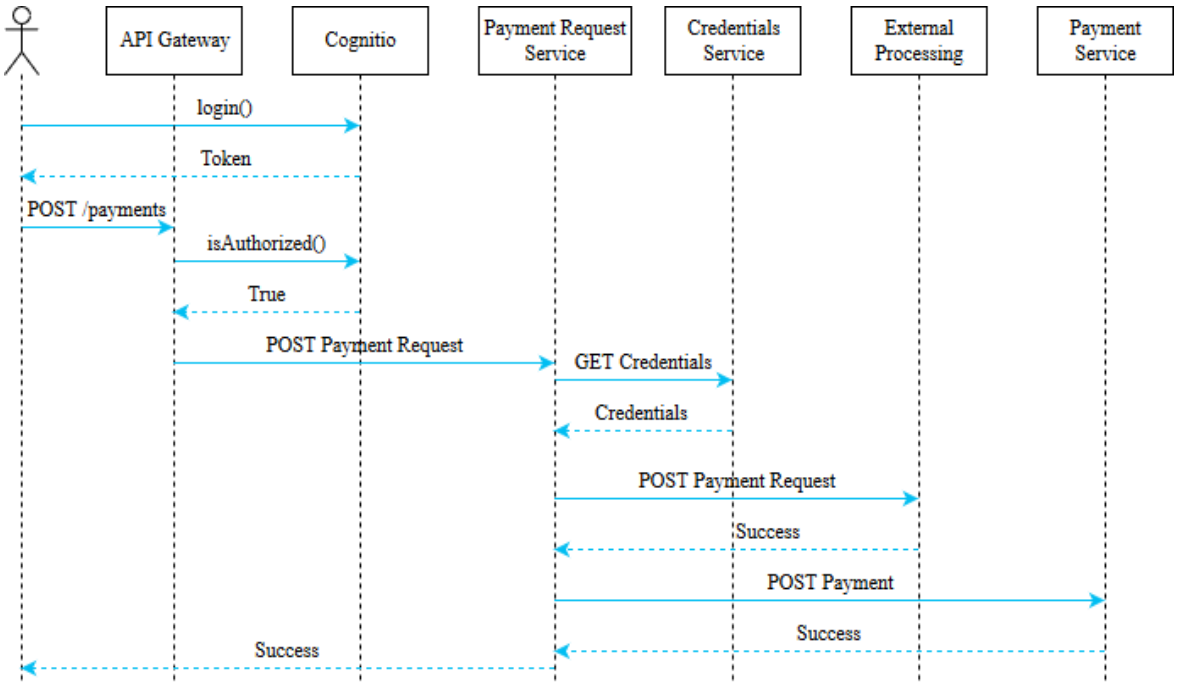

Figure 12. Sequence diagram payment process in AWS

7.2.3 A microservice in AWS

The microservices are developed locally with JavaScript using the Express library as a simple REST API endpoint. The microservices are running in a Node.js environment and are built as Docker containers images. After testing and finishing the images locally, the images are pushed with a tag to a repository in Amazon Elastic Container Registry (ECR). The ECR oversees the registering and the storing of different Docker image versions, which can be identified by their tags in different repositories. In the case that a microservice gets changed or new features are added, a new image needs to be built and pushed to ECR by overwriting a tag or creating a new tag.

Images stored in ECR can be used in a task definition for running a Docker container in the Amazon Elastic Container System (ECS). A task definition can be configured in the Amazon ECS console or with a JSON configuration file and the AWS CLI. Furthermore, port mappings are defined in the ECS task definition. A port mapping is the description, in which a container port is open for listening for connections. The container port is configured in the JavaScript Express code to listen on port 80. Additionally, environment variables can be set in the configuration of a task definition, for example to set the address of a Network Load Balancer of another microservice for inter service communication. Furthermore, the task size is chosen in form of a memory size and a CPU size. In the case, a 0.5 GB memory size and a 0.25 vCPU (virtual CPU) task size are used.

A microservice, which is illustrated in detail in Figure 13, is created as a service with a task definition in the Amazon ECS console. A task definition has a launch type, which is in the case AWS Fargate. AWS Fargate deploys the container instances of the task to a chosen cluster, which is configured in a service. A cluster provides virtual resources, on which the task instances of a service run. The resources provisioning to the task instances are managed by AWS Fargate. Furthermore, a service in ECS can be updated with a new revision of the task definition. In addition, in a service the desired, minimum, and maximum number of task instances can be defined.

AWS Cloud Watch monitors the task instances of a service on different Cloud Watch Metrics for example the CPU or the memory utilization. A service can have policies configured

with Cloud Watch Alarms, which are triggering the AWS Auto Scaling of the task instances. A policy should be configured so it has enough time to adjust the number of task instances according to a load change. For example, in the case a policy of a memory utilization greater than 50 % is used to scale a new task instance. The up- and downscaling of container instances in the ECS takes several minutes from initiation to a healthy state. The scaling of instances happens in steps with a configurable cooldown period in between. This leads to a service usually needing to be overprovisioned to adjust in time for a possible upcoming higher load.

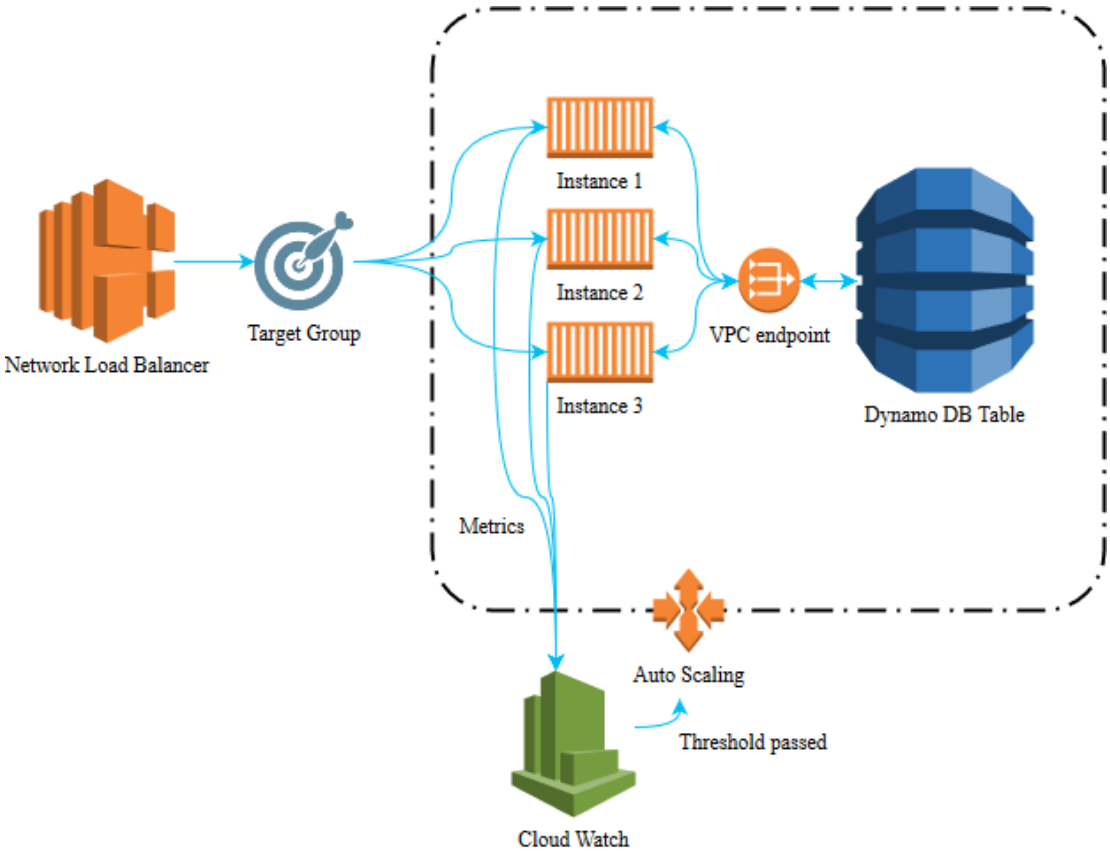

Figure 13. Microservice Auto Scaling group

A target group of a load balancer can be defined in a service as well. In the architecture, a Network Load Balancer with a target group is defined for each microservice. The task instances register themselves to the target group managed by the Network Load Balancer. The Network Load Balancer sends health requests periodically to the task instances of the target group to determine the healthy targets. Requests arrive for the microservice at the Network

Load Balancer and the Network Load Balancer distributes them only to healthy targets. The Network Load Balancer selects a target instance with a flow hash algorithm, which directs all calls from a single client to the same target as long as the connection exists. An unhealthy target does not get any requests until a health check succeed. If a target instance is crashed or unhealthy for a longer time, it gets shutdown and a new instance is created for it. If there is no healthy target in a target group, then a request gets rejected with an error code. The Network Load Balancer receives requests over the Transmission Control Protocol (TCP) and sends TCP requests to the target group. The load balancer forwards the request without opening or changing the HTTP part of the request.

A microservice can interact with a database during processing a request. In the case, the NoSQL database DynamoDB is chosen as it scales very well and has a good performance on simple read and write operations. For each microservice that needs data persistence a separate DynamoDB table is created. Different instances of a microservice share a DynamoDB table. In the case, a credentials and a payments table is created for the different microservices. A DynamoDB table scales accordingly to the read or write capacity units separately configured with AWS Auto Scaling. Furthermore, the DynamoDB instance is configured with a policy to only get accessed via VPC endpoints. Hence, the tables can only be accessed by a pre-defined microservice, which provides a high reliability that the user data is secured.

7.2.4 Assessment

The availability in the microservice architecture in AWS is assessed on several different parts that could be single points of failures in this architecture. AWS ensures for ECS instances provided with AWS Fargate a SLA of 99,99 % uptime and availability (AWS, 2017), which corresponds to an allowable downtime of around five minutes in a month. If the availability is not reached, a service credit will be provided for the customers. AWS offers different isolated availability zones within a region to provide a higher availability by replicating services in different zones (AWS, 2018k). Different task instances in ECS can be deployed to different availability zones. There is currently no SLA defined for DynamoDB, but the data in DynamoDB is replicated within a region to three different availability zones

to achieve a good availability and a high uptime (AWS, 2018c). Furthermore, there is no SLA defined for the AWS Network Load Balancer and the Amazon API Gateway. The clients and microservices must be configured to handle possible errors in the connection process with the Network Load Balancer and the API Gateway. In summary, the implemented microservice architecture provides a good availability with well-proven reliable services and by provisioning the services in different availability zones. However, the architecture includes several single points of failures like the load balancers and the gateway, which would shut down the application in the case of a failure.

The scalability is also assessed on different parts of the architecture. The scaling of ECS container task instances takes several minutes after the desired task amount has changed by a Cloud Watch alarm or a manual change. That is why the microservices need to be overprovisioned to serve all requests with at least one task always running. Hence, the resource utilization is not optimal, because the provisioned resources are in general always higher than the actual demand of resources. On the other hand, in a worse scenario of a sudden and unexpected load a microservice could have a resource shortage as the scaling is too slow to adjust to the load. The DynamoDB scales with AWS Auto Scaling, which works fine for stable periodic changes in the load. A sudden unexpected load could mean the throttling of the database requests, which means that the client would receive an error from its request. Amazon API Gateway and AWS Network Load Balancer are scaled automatically configured by AWS within default throttling limits. AWS ECS has a default limit of number of container instances per cluster. AWS DynamoDB has a default limit of 40,000 read capacity and write capacity per table in the North Virginia Region. A read capacity corresponds to two eventually consistent read operations per second and a write capacity corresponds to a write operation per second. Overall the default limits in DynamoDB are 80,000 read capacity and write capacity in an account in the North Virginia region. As a result, two microservices could block the entire database in a region with these default limits. All these limits can be increased by making a support request to AWS. However, a cloud consumer could face these default limits unexpectedly.

The reliability in the microservice architecture is dependent on the availability of the system and its performance. In general, the performance of the payment processing is fast, because

of a slow scaling latency the microservices have usually more resources available than needed for a load. Only sudden burst loads can lead to a lower performance, because the scaling of additional task instances need some time to adjust correctly to the demand. The user data is secured in this architecture through token authorization, and the DynamoDB database is only accessible via microservices within the VPC. Hence, clients cannot directly connect to the database.

The amount of needed resources for this architecture are high compared to the Serverless architecture, because a high work load is needed for the setup and the configuration of the architecture and the communication within this architecture. Additionally, the development and extension of a microservice is time intensive, because the service must be built as a Docker container and deployed to the Amazon Elastic Container Registry for every change. Furthermore, more computing resources are needed to handle rapidly changing user amounts, because microservices must be overprovisioned and run on idle times.

7.2.5 Cost estimation[6]

The cost estimation for the microservices architecture in AWS is in Table 3 for 100 million payment requests in a month with the usage of unchanging free tiers. The same payment JSON of 200 Bytes is used as in the Firebase cost estimation. For the DynamoDB it is assumed that 40 read operations and 40 write operations per second throughput capacity is needed with an eventually consistency for reading data. The Elastic Container Service is providing four tasks of 512 MB memory and 0.25 vCPU with Fargate and the assumption that there is no need to scale them. For Fargate the prices of $0.0506 per hour per vCPU and $0.0127 per hour per GB memory are used (AWS, 2018d). In this architecture, there are four network load balancers used, which have 40 active connections with a 1KB bandwidth. The total costs of around $496 for this architecture is nearly double as high as the Serverless architecture in Firebase. However, the costs are still low in relation to the high number of payment requests.

[6] Usage of AWS Calculator in February 2018 available on https://calculator.s3.amazonaws.com/index.html

Costs for DynamoDB	in $
18.6 GB stored Dataset (25 GB free Tier)	0.00
Provision Throughput Capacity 40 read operations/s & 40 write operations/s	7.90
Costs for Elastic Container Service (Fargate)	
720 h * $0.0127 per hour * 0.5 GB * 4 tasks	18.30
720 h * $0.0506 per hour * 0.25 vCPU * 4 tasks	36.40
Data Transfer Banking Service ~ 95 GB	8.50
Costs for Network Load Balancers	
4 NLB 40 active connections 1 KB bandwidth	66.50
Costs for API Gateway	
100 million API Calls ($3.50/million)	350.00
Data Transfer Response ~ 95 GB ($0.09/GB)	8.55
Total estimated costs	**496.15**
Price per payment	**0.00000496**

Table 3. Cost estimation AWS implementation

7.2.6 Drawbacks and possible improvements

The inter service communication is configured with synchronous calls between the microservices. This means a microservice must wait for another microservice with an open connection to finish its task. For overcoming this bottleneck of open connections, one solution could be to use message queues in between the microservices to encapsulate the communication. In this way the communication would be asynchronous. This joint architecture would only work, if each message includes all the parameters needed for further tasks to make each step independent from each other.

The scaling might not be efficient and fast enough for sudden load bursts in the number of payment requests. The scaling of container instances takes several minutes and happens in steps. AWS is offering a fast and automatic scaling Serverless Function as a Service called AWS Lambda. The Docker containers in ECS could be replaced with Lambda functions, which would be able to better adjust to sudden load bursts and AWS would handle the scaling of Lambda functions automatically with the correct amount of resources in a Serverless way. Furthermore, the development and deployment would be easier with AWS Lambda instead of building a microservice in a Docker container.

8 Discussion

The Serverless architecture and microservice architecture are implemented for the case of a mobile payment application with the possibility to have many users. The objectives of availability, scalability, reliability, and needed resources are used for the assessment of the architecture approaches. Overall, both architecture implementations are suitable for the case of the mobile payment application and other applications in the domain based on the analysis. However, in both solutions drawbacks can be found, which should be addressed by the cloud provider to enhance the quality of their services in future. Furthermore, the cloud consumer can develop both architectures further to improve the objectives.

The availability of an architecture is difficult to assess as both implementations are generally available services, but with the possibility of an unavailable service caused by cloud outage or a single service failure. Therefore, it must be identified how the implementations are prone to failures. The microservice architecture in AWS has the API Gateway and the Network Load Balancers as single point of failures. The Serverless architecture in Firebase has the Realtime Database as a single point of failure. The API Gateway and the Network Load Balancer in the AWS architecture are not backed up with a SLA, but they have been proven reliable by the usage of many cloud consumers in the AWS environment. On the other hand, the Realtime Database in the Serverless architecture has a SLA of 99.95 % uptime. The availability in the computing logic part is ensured by AWS ECS with a SLA of 99.99 % uptime unlike Cloud Function for Firebase where no SLA is defined for the beta product. The costs for using the logic part are small in both cases. It could be discussed that having a SLA and a compensation will not help in the case of a failure. However, if a cloud provider gives a SLA, the cloud provider is more confident that the offered cloud service is available.

Different AWS ECS instances are deployed to different availability zones within a region. Therefore, if one availability zone goes down, the microservice can be served in another zone. Additionally, the DynamoDB is a replicated database to different availability zones in the microservice architecture, but it has no SLA provided. In summary, it is difficult to say which architecture solution will have the better availability overall, because different parts of each architectures have an availability ensured with a SLA and other parts are without

this assurance. Nevertheless, the deployment and replication to different availability zones in the microservice architecture in AWS favors this architecture over the Serverless architecture in Firebase.

The scalability differs in the scaling latency of the architectures. The Serverless architecture in Firebase has a better scaling latency by providing new instances or removing instances quickly according to the load. In contrast, the microservices architecture in AWS takes several minutes to turn a new instance on or off to adjust to the current load. This leads to the Firebase architecture having a better resource utilization than the AWS architecture by providing the correct amount of logic resources to the current load. However, the scalability of the Firebase architecture has scalability limits of invocations per second for Cloud Functions for Firebase. There is currently a strict limit of 1,000 payments per second in the case of the payment processing. Furthermore, the Firebase Realtime Database has a connection limit of 100,000 to an instance, which could be problematic, because the Realtime Database is the focal point of the Serverless architecture in Firebase and all different kinds of requests are running over the Realtime Database. On the other hand, the DynamoDB in the AWS architecture has a throughput capacity, which can be increased or decreased through AWS Auto Scaling according to the load to a certain default limit. This limit can be increased with a support request. To sum up, the scalability is better in the Serverless architecture in Firebase until a certain limit is reached, after that the microservice architecture in AWS is in favor.

The reliability of the service is assessed in the performance of the payment processing and the data security of personal data. A more reliable performance for the payment processing is given by the microservice architectures in AWS, because there is always a running ECS task container instance that can immediately process a payment process. In the Serverless architecture, the payment processing function could encounter a cold start if it is not regularly triggered and therefore it must start up, which increases the latency. In an application with many users, this should not be a problem as consequently payment requests are processed. However, the Serverless architecture in Firebase has a better performance in the case of a sudden burst load as the scaling adjusts faster and thus more resources are available to handle the load.

The data is secured in the AWS microservice approach in a virtual private cloud of AWS. Only the microservice instances themselves have access to a certain database table. On the contrary, in the Serverless architecture in Firebase clients are directly connected to the Realtime Database instance. The Realtime Database has rules definitions that are controlling the data access. However, the user expectations of data security are better expressed in the microservice architecture in AWS and it is the more reliable service currently in the performance.

Other factors for deciding between the two solutions are the amount of work and the costs for implementing a solution. The Serverless architecture in Firebase is quickly planned and set up without the need to configure any server. In contrast, the microservice architecture needs more detailed planning and different services must be set up and configured separately. The development of containers and the setup of the container environment takes additional time in contrast to the development of cloud functions. A developer can concentrate himself more on the application code with the Serverless architecture in Firebase. The estimated costs for processing 100 million payment requests are almost double as high for the microservice architecture in AWS. However, both cost estimations show a negligible price per payment for the high number of payments. The difference is more visible with a low number of payments, where the $120 fixed costs for the microservice architecture in AWS stands against no fixed costs of the Serverless architecture in Firebase. Summing up the Serverless approach is an advisable architecture for the beginning phase of an application with a lower initial amount of work and no fixed costs for the environment.

The requirements of a company must be considered to decide for an architectural cloud backend solution. In Table 4 the assessment of the solution is summarized. In the case of the mobile payment application, the Serverless architecture in Firebase is a good starting point to have an architecture that supports millions of users. However, if the application has beyond millions of users, it could be considered to change the architecture to a more reliable microservice architecture in AWS, because the Serverless architecture in Firebase has user and configuration limitations. Furthermore, different parts of other architectures could be added to the microservice architecture in AWS like messages queues between microservices to have an asynchronous inter service communication. Additionally, container instances

could be replaced with Serverless Lambda functions to reduce the scaling latency and deployment time. This leads to the future work in this field, which has improvement possibilities in the architecture design and the implementation.

	Serverless architecture in Google Firebase	Microservice architecture in Amazon Web Services
Availability	Realtime Database with SLA	SLA for ECS, availability zones for ECS and DynamoDB
Scalability	Low scaling latency, better resource utilization, scalability limits	High scaling latency
Reliability	Cold starts of functions	Problem with sudden burst loads
Needed resources	Low	Setup and configuration of the architecture
Price per payment	$ 0.00000297	$ 0.00000496

Table 4. Assessment of the implemented cloud architectures

8.1 Future work

After this study is concluded, cloud providers should improve their offered cloud services in order to be more attractive for companies to use their services. Both practical implementations are currently hosted and only available in a US region. Hence, cloud providers should offer their services in different regions around the globe to improve the performance and fulfill law requirements for certain countries. Furthermore, cloud consumers could build then

multi-region architectures, which would have a higher availability and therefore be less prone to failures. Additionally, more assurance on the availability of the services could be given to the cloud consumer, if a cloud provider would ensure a SLA for all offered cloud services. Current limitations in cloud services should be increased or removed to allow cloud consumers a higher scalability of their implementations. Furthermore, a cloud provider could provide more boilerplates to make the setup of an environment easier for the cloud consumer for complex architectures like the microservice architecture.

Research is another approach to improve the work in this area for companies and institutions. There is a need for constant research in this field, as it is changing constantly, solutions are being updated, and new service solutions and architectures are being offered. Especially, the research on designing a cloud architecture for many users is little. More research in this area is important and necessary, because for example mobile applications receive increasing importance and users. The scalability of different cloud solutions has limits or certain bottlenecks, which can be broken with an unexpected increased user amount. Those bottlenecks need to be identified and new services should be developed to be prepared for upcoming huge user amounts.

The underlying research method of this study is a development research (Nunamaker Jr et al., 1990), which is a continuous and iterative process, which means the implemented architecture for an application with many users should be constantly updated and evaluated. For this, public cloud providers could update their services and provide new services, which could be useful for the application to improve the objectives. Hence, a cloud architecture for an application with many users is never finished and can always be improved.

9 Conclusion

In this thesis, general concepts of cloud computing are introduced, and a definition of cloud computing is given. Additionally, virtualization and database concepts are discussed in relation to cloud computing. The case of a mobile payment application is presented, and its objectives of availability, scalability, reliability, and needed resources are defined. The tier-based architecture, the message queue architecture, the microservice architecture and the Serverless architecture are explained and are compared to each other according to the objectives.

The thesis shows that different architectural solutions have their pros and cons to different objectives of different actors in this area. This confirms the statement of Rimal et al. that there is no single architectural method for designing a cloud backend (2011). Instead, the cloud architecture of a case needs always to be designed on the objectives, constraints, and requirements of a case. In the comparison, it is identified that different architectures approaches are developing from the tier-based architecture with a lot of configuration possibilities to the Serverless architecture without any configuration at all. The modern approaches of a microservice architecture and a Serverless architecture are assessed to be the most promising architectures solutions for the case of the mobile payment application and other applications in the same domain, because of their partitioning in small computing instances and thus a better scalability according to the demand of a certain functionality.

The Serverless architecture in Firebase and the microservice architecture in AWS are practically implemented and assessed. The assessment of the Serverless architecture shows that the implementation has an excellent scaling latency, but the scalability has certain limitations. These limitations could change in Firebase environment or are not present in another Serverless environment. In contrast, the microservice architecture implementation in AWS is examined as a more reliable solution, because of a better performance in the payment processing, data security of personal data and availability in the different services. However, the microservice architecture has fixed costs and requires initially more work in the setup of the environment.

It therefore becomes clear that there is no single answer to the research question of how to design a cloud architecture for an application with many users. The Serverless architecture is a good implementation for the case until it receives user amounts of several millions or billions. Then it could be considered to change the architecture to a more complex architecture like the microservice architecture with more configuration possibilities and less limitations.

This study has thus provided some interesting considerations and answers to the central research question. The thesis gives a collection of different cloud architecture approaches and assesses them on different objectives. Other implementations in future could profit from the findings of this thesis in their architecture design phase and in the actual implementation. Further research in this field should aim on upcoming new services, updated solutions, and their capabilities to improve an architecture or to create a new architecture type for handling an increasing and unexpected number of users for applications in future.

References

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., . . . Stoica, I. (2010). A view of cloud computing. *Communications of the ACM, 53*(4), 50-58.

AWS. (2017). Amazon compute service level agreement. Retrieved from https://aws.amazon.com/ec2/sla/

AWS. (2018a). Amazon API gateway FAQs. Retrieved from https://aws.amazon.com/api-gateway/faqs/

AWS. (2018b). Amazon CloudWatch - cloud & network monitoring services. Retrieved from https://aws.amazon.com/cloudwatch/

AWS. (2018c). Amazon DynamoDB FAQs – amazon web services (AWS). Retrieved from https://aws.amazon.com/dynamodb/faqs/

AWS. (2018d). Amazon ECS - run containerized applications in production. Retrieved from https://aws.amazon.com/ecs/

AWS. (2018e). Amazon virtual private cloud (VPC). Retrieved from https://aws.amazon.com/vpc/

AWS. (2018f). AWS | amazon elastic container registry | docker registry. Retrieved from https://aws.amazon.com/ecr/

AWS. (2018g). AWS auto scaling. Retrieved from https://aws.amazon.com/autoscaling/

AWS. (2018h). AWS fargate - run containers without having to manage servers or clusters. Retrieved from https://aws.amazon.com/fargate/

AWS. (2018i). AWS lambda – FAQs. Retrieved from https://aws.amazon.com/lambda/faqs/

AWS. (2018j). Elastic load balancing product details. Retrieved from https://aws.amazon.com/elasticloadbalancing/details/

AWS. (2018k). Regions and availability zones - amazon elastic compute cloud. Retrieved from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2015). Migrating to cloud-native architectures using microservices: An experience report. Paper presented at the *European Conference on Service-Oriented and Cloud Computing*, 201-215.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., . . . Slominski, A. (2017). Serverless computing: Current trends and open problems. *arXiv Preprint arXiv:1706.03178*,

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. Paper presented at the *Proceedings of the 2nd International Workshop on Software and Performance*, 195-203.

Brewer, E. A. (2000). Towards robust distributed systems. Paper presented at the *Podc*, *7*

Brown, K., Craig, G., Hester, G., Amsden, J., Pitt, D., Jakab, P. M., . . . Weitzel, M. (2003). *Enterprise java programming with ibm websphere* Addison-Wesley Professional.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *Acm Sigmod Record, 39*(4), 12-27.

Christensen, J. H. (2009). Using RESTful web-services and cloud computing to create next generation mobile applications. Paper presented at the *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 627-634.

Constine, J. (2017). Facebook messenger hits 1.2 billion monthly users, up from 1B in july. Retrieved from http://social.techcrunch.com/2017/04/12/messenger/

Curino, C., Jones, E. P., Popa, R. A., Malviya, N., Wu, E., Madden, S., . . . Zeldovich, N. (2011). Relational cloud: A database-as-a-service for the cloud.

Docker. (2015). What is docker? Retrieved from https://www.docker.com/what-docker

Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., & Hu, B. (2015). Everything as a service (XaaS) on the cloud: Origins, current and future trends. Paper presented at the *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference On*, 621-628.

Express. (2018). Express - node.js web application framework. Retrieved from https://expressjs.com/

Firebase. (2018a). Choose a database: Cloud firestore or realtime database. Retrieved from https://firebase.google.com/docs/database/rtdb-vs-firestore

Firebase. (2018b). Cloud firestore. Retrieved from https://firebase.google.com/docs/firestore/

Firebase. (2018c). Cloud functions for firebase. Retrieved from https://firebase.google.com/docs/functions/

Firebase. (2018d). Firebase authentication | simple, free multi-platform sign-in. Retrieved from https://firebase.google.com/products/auth/

Firebase. (2018e). Firebase realtime database. Retrieved from https://firebase.google.com/docs/database/

Firebase. (2018f). Service level agreement for hosting and realtime database. Retrieved from https://firebase.google.com/terms/service-level-agreement

Fowler, M. (2002). *Patterns of enterprise application architecture* Addison-Wesley Longman Publishing Co., Inc.

Fowler, M., & Lewis, J. (2014). Microservices. Retrieved from http://martinfowler.com/articles/microservices.html

Fox, G. C., Ishakian, V., Muthusamy, V., & Slominski, A. (2017). Status of serverless computing and function-as-a-service (FaaS) in industry and research. *arXiv Preprint arXiv:1708.08028,*

Google Cloud Platform. (2017). Pricing | cloud functions documentation. Retrieved from https://cloud.google.com/functions/pricing

Google Cloud Platform. (2018). Quotas | cloud functions documentation. Retrieved from https://cloud.google.com/functions/quotas

Grozev, N., & Buyya, R. (2013). Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments. *The Computer Journal*, *58*(1), 1-22.

Gunarathne, T., Wu, T., Choi, J. Y., Bae, S., & Qiu, J. (2011). Cloud computing paradigms for pleasingly parallel biomedical applications. *Concurrency and Computation: Practice and Experience, 23*(17), 2338-2354.

Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on NoSQL database. Paper presented at the *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference On,* 363-366.

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

He, C., Fan, X., & Li, Y. (2013). Toward ubiquitous healthcare services with a novel efficient cloud platform. *IEEE Transactions on Biomedical Engineering*, *60*(1), 230-234.

Heuveldop, N. (2017). Ericsson mobility report. *Ericsson AB, Technol.Emerg.Business, Stockholm, Sweden, Tech.Rep.EAB-17, 5964*

Höfer, C. N., & Karagiannis, G. (2011). Cloud computing services: Taxonomy and comparison. *Journal of Internet Services and Applications, 2*(2), 81-94.

Homer, A., Sharp, J., Brader, L., Narumoto, M., & Swanson, T. (2014). *Cloud design patterns: Prescriptive architecture guidance for cloud applications* Microsoft patterns & practices.

Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. Paper presented at the *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances In,* 342-346.

Keahey, K., Armstrong, P., Bresnahan, J., LaBissoniere, D., & Riteau, P. (2012). Infrastructure outsourcing in multi-cloud environment. Paper presented at the *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit,* 33-38.

Khiyaita, A., El Bakkali, H., Zbakh, M., & El Kettani, D. (2012). Load balancing cloud computing: State of art. Paper presented at the *Network Security and Systems (JNS2), 2012 National Days Of,* 106-109.

Kusic, D., Kephart, J. O., Hanson, J. E., Kandasamy, N., & Jiang, G. (2009). Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing, 12*(1), 1-15.

Leavitt, N. (2010). Will NoSQL databases live up to their promise? *Computer, 43*(2)

Li, A., Yang, X., Kandula, S., & Zhang, M. (2010). CloudCmp: Comparing public cloud providers. Paper presented at the *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement,* 1-14.

Li, Y., & Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. Paper presented at the *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference On,* 15-19.

Lu, W., Jackson, J., & Barga, R. (2010). AzureBlast: A case study of developing science applications on the cloud. Paper presented at the *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing,* 413-420.

Lu, Y., Yang, S., Chau, P. Y., & Cao, Y. (2011). Dynamics between the trust transfer process and intention to use mobile payment services: A cross-environment perspective. *Information & Management, 48*(8), 393-403.

Malawski, M. (2016). Towards serverless execution of scientific workflows-HyperFlow case study. Paper presented at the *Works@ Sc,* 25-33.

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing—The business perspective. *Decision Support Systems, 51*(1), 176-189.

McGrath, G., & Brenner, P. R. (2017). Serverless computing: Design, implementation, and performance. Paper presented at the *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference On,* 405-410.

Mell, P., & Grance, T. (2011). The NIST definition of cloud computing.

Namiot, D., & Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies, 2*(9), 24-27.

Newman, S. (2015). *Building microservices: Designing fine-grained systems* " O'Reilly Media, Inc.".

Nunamaker Jr, J. F., Chen, M., & Purdin, T. D. (1990). Systems development in information systems research. *Journal of Management Information Systems, 7*(3), 89-106.

Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., & Abramov, J. (2011). Security issues in nosql databases. Paper presented at the *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference On*, 541-547.

Pokorny, J. (2013). NoSQL databases: A step to database scalability in web environment. *International Journal of Web Information Systems, 9*(1), 69-82.

Pritchett, D. (2008). Base: An acid alternative. *Queue, 6*(3), 48-55.

Rahimi, M. R., Venkatasubramanian, N., Mehrotra, S., & Vasilakos, A. V. (2012). MAPCloud: Mobile applications on an elastic and scalable 2-tier cloud architecture. Paper presented at the *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference On*, 83-90.

Rimal, B. P., Choi, E., & Lumb, I. (2009). A taxonomy and survey of cloud computing systems. *Ncm, 9*, 44-51.

Rimal, B. P., Jukan, A., Katsaros, D., & Goeleven, Y. (2011). Architectural requirements for cloud computing systems: An enterprise cloud approach. *Journal of Grid Computing, 9*(1), 3-26.

Roberts, M. (2016). Serverless architectures. Retrieved from https://martinfowler.com/articles/serverless.html

Sareen, P. (2013). Cloud computing: Types, architecture, applications, concerns, virtualization and role of it governance in cloud. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(3)

Satzger, B., Hummer, W., Inzinger, C., Leitner, P., & Dustdar, S. (2013). Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing, 17*(1), 69-73.

Schierz, P. G., Schilke, O., & Wirtz, B. W. (2010). Understanding consumer acceptance of mobile payment services: An empirical analysis. *Electronic Commerce Research and Applications, 9*(3), 209-216.

Sparks, D. (2017). How many users does WhatsApp have? Retrieved from https://www.fool.com/investing/2017/04/06/how-many-users-does-whatsapp-have.aspx

Spillner, J., Mateos, C., & Monge, D. A. (2017). FaaSter, better, cheaper: The prospect of serverless scientific computing and HPC. Paper presented at the *4th Latin American Conference on High Performance Computing (CARLA). to Appear*,

Stone, L. (2016). Bringing pokémon GO to life on google cloud. Retrieved from https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html

Stubbs, J., Moreira, W., & Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. Paper presented at the *Science Gateways (IWSG), 2015 7th International Workshop On*, 34-39.

Synergy Research Group. (2017). Cloud market keeps growing at over 40%; amazon still increases its share. Retrieved from https://www.srgresearch.com/articles/cloud-market-keeps-growing-over-40-amazon-still-increases-share

Tauro, C. J., Aravindh, S., & Shreeharsha, A. B. (2012). Comparative study of the new generation, agile, scalable, high performance NOSQL databases. *International Journal of Computer Applications, 48*(20), 1-4.

Thönes, J. (2015). Microservices. *IEEE Software, 32*(1), 116.

Toeroe, M., & Tam, F. (2012). *Service availability: Principles and practice* John Wiley & Sons.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., & Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. Paper presented at the *ACM SIGMETRICS Performance Evaluation Review, 33*(1) 291-302.

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review, 41*(1), 45-52.

Wasson, M. (2017). Web-queue-worker architecture style. Retrieved from https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/web-queue-worker

Wilder, B. (2012). *Cloud architecture patterns: Using microsoft azure* " O'Reilly Media, Inc.".

Wolf, O. (2018). Purposes of the serverless architecture style. Retrieved from https://specify.io/concepts/serverless-baas-faas

Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., & De Rose, C. A. (2013). Performance evaluation of container-based virtualization for high performance computing environments. Paper presented at the *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference On,* 233-240.

Younge, A. J., Henschel, R., Brown, J. T., Von Laszewski, G., Qiu, J., & Fox, G. C. (2011). Analysis of virtualization technologies for high performance computing environments. Paper presented at the *Cloud Computing (CLOUD), 2011 IEEE International Conference On,* 9-16.

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