BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

CLOUD COMPUTING APPLICATION DESIGN PATTERNS
NÁVRHOVÉ VZORY V CLOUD COMPUTING APLIKACÍCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**Bc. MATEJ KOLESÁR**
SUPERVISOR **VEDOUCÍ PRÁCE**	**RNDr. MAREK RYCHLÝ, Ph.D.**

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

143978

Institut:	Department of Information Systems (UIFS)
Student:	**Kolesár Matej, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Cloud Computing Application Design Patterns**
Category:	Software Engineering
Academic year:	2022/23

Assignment:

1. Become familiar with cloud computing environments, types of applications for these environments and how to provide services. Explore application development capabilities for cloud computing environments and the resources offered by their providers.
2. Learn how to design cloud computing applications and the most commonly used design patterns.
3. Design a sample application, which could be used to demonstrate design patterns for cloud computing applications and which could be deployed at available providers.
4. After consulting with the manager, implement the proposed application and describe the sample use of the mentioned design patterns.
5. Thoroughly document the design procedure and implementation of the application parts where individual design patterns were applied: describe each pattern, assumptions and consequences of its application, the specific application procedure, result and other possibilities. Verify in practice that the pattern had the expected benefits (e.g., better application scalability).
6. Evaluate the results and publish them as open-source.

Literature:

- D. Comer: The cloud computing book: the future of computing explained. First edition. ISBN 978-0-367-70680-7
- Ch. M. Moyer: Building applications in the cloud: concepts, patterns, and projects. Upper Saddle River, NJ, Addison-Wesley, 2011. ISBN 978-0-321-72020-7
- Ch. Fehling, F. Leynman, F. Rette, W. Schupeck and P. Arbitter: Cloud Computing Patterns. Springer-Verlag Wien, 2014. ISBN 978-3-7091-1568-8

Requirements for the semestral defence:
Items 1, 2, and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	24.10.2022

Abstract

This thesis aims to demonstrate available cloud patterns which solve existing problems that are experienced in the cloud environment. Various cloud patterns are first analysed from a high-level view and then further studied on a lower component level. These components and architectures provide certain solutions depending on the use case of the application. A demo application is designed to showcase these design patterns and how they behave. The implementation is done using kubernetes and it is deployed to AWS. The chosen architecture uses microservices. The application consists of 2 designs. The first one shows the AWS advantages and the second one can be deployed on private clouds but also on AWS. At the end, experiments are performed that verify whether the used patterns had the expected results.

Abstrakt

Cieľom tejto práce je demonštrovať existujúce cloudove vzory, ktoré riešia problémy v cloud prostrediach. Rôzne cloudove vzory sú analyzované najprv na vyššej úrovni z pohľadu architektúry aplikácie a následne aj na nižších úrovniach pre jednotlivé komponenty. Tieto architektúry a komponenty poskytujú výhodu v istých situáciách a záleží na správaní aplikácie, ako veľmi zjednodušia a zlepšia využívanie cloud prostredia. Je navrhnutá demo aplikácia, ktorá ma 2 návrhy. Prvý návrh používa servisy, ktore vyzdvihuju výhody AWS a druhý návrh možno nasadiť v súkromných cloudoch ale aj na AWS. Aplikácia je nasadzovaná pomocou kubernetov a používa microservisy ako zvolenú architektúru. Po nasadení sú nad aplikáciou urobené experimenty, ktoré slúžia na overenie použitých vzorov a či mali očakávané dopady na aplikáciu.

Keywords

Cloud, Cloud computing, Cloud design patterns, Microservice Architecture, Automation, Kubernetes, Java

Klíčová slova

Cloud, Cloud computing, Cloud Navrhove vzory, Architektúra microservis, Automatizacia, Kubernetes, Java

Reference

KOLESÁR, Matej. *Cloud Computing Application Design Patterns.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Cloud computing sa stal neoddeliteľnou súčasťou modernej počítačovej infraštruktúry vďaka mnohým výhodám, ako je flexibilita, škálovateľnosť a nákladová efektívnosť. S rastúcou zložitosťou cloudových prostredí sa však objavilo niekoľko problémov, ktoré je potrebné riešiť. V tejto súvislosti je cieľom tejto práce poskytnúť analýzu dostupných vzorov cloudu, ktoré môžu pomôcť tieto problémy riešiť.

Práca sa začína prehľadom existujúcich cloudových vzorov, po ktorom nasleduje analýza ich funkčnosti a použiteľnosti na vysokej úrovni. Následne sa práca hlbšie venuje nižším komponentom vzorov a skúma, ako ich možno využiť v rôznych prípadoch použitia. Vzory sú analyzované od návrhových vzorov pre architektúru aplikácie. Súčasťou práce je taktiež analýza rôznych architektúr a výhod, ktoré poskytujú v určitých situáciách. Následne sú analyzované detailnejšie komponenty, ktoré sú väčšinou zamerané na jeden konkrétny problém a v návrhu celej aplikácie sa opakujú častejšie. Jeden z týchto vzorov je kompotent na vyrovnávanie záťaže v cloude, ktorý zabezpečuje správnu distribúciu správ medzi inštanciami servisov. Okrem vyrovnávania záťaže sú analyzované aj komponenty, ktoré zabezpečujú lepšiu škálovateľnosť a komunikáciu medzi servismi. Taktiež sú analyzované štýly, ktoré môžu nastať a ako je možné jednotlivé typy využiť. Konečným cieľom je ukázať potenciál cloudových vzorov pri riešení bežných problémov vyskytujúcich sa v prostredí cloudu. Následne je navrhnutá aplikácia, ktorej účelom bude aplikovať tieto vzory. Návrh aplikácie je rozoberaný po jednotlivých funkčných častiach aplikácie a jednotlivé časti sú analyzované do detailu. Je popísané aké je predpokladané správanie aplikácie a aké dáta sú očakávané.

Na implementáciu jednotlivý microservisov a business logiky je použitý jazyk Java a framework spring boot. Tento framework poskytuje množstvo funkcií vrátane ľahko použiteľných rozhraní API, efektívneho smerovania požiadaviek a vyrovnávania záťaže, ktoré sú nevyhnutné na budovanie škálovateľných a odolných mikroslužieb. Vyspelý ekosystém Java navyše ponúka širokú škálu nástrojov a knižníc, ktoré možno využiť na zlepšenie vývoja a nasadenia mikroslužieb v cloude. Niektoré z týchto nástrojov zahŕňajú Docker pre kontajnerizáciu a Kubernetes pre orchestráciu, ktoré boli použité v tejto práci. Okrem toho kompatibilita Java s populárnymi cloudovými platformami, ako sú Amazon Web Services (AWS), Microsoft Azure a Google Cloud Platform (GCP), umožňuje jednoduché nasadenie a škálovanie mikroslužieb. Na zabezpečenie efektívnej a spoľahlivej prevádzky mikroslužieb v cloude musia vývojári Java zvážiť faktory, ako je výkon, bezpečnosť a monitorovanie. Dá sa to dosiahnuť použitím profilovacích nástrojov, bezpečnostných knižníc a protokolovacích a monitorovacích framework.

Na demonštráciu účinnosti vybraných vzorov je implementovaná aplikácia s využitím populárneho nástroja na orchestráciu kontajnerov Kubernetes a nasadená na jednej z popredných cloudových platforiem Amazon Web Services (AWS). Zvolenou architektúrou pre demonštračnú aplikáciu sú mikroslužby, ktoré v porovnaní s monolitickou architektúrou poskytujú lepšiu škálovateľnosť, odolnosť a flexibilitu. Demonštračná aplikácia je navrhnutá tak, aby prezentovala dva rôzne návrhy: jeden, ktorý vyzdvihuje výhody AWS, a druhý, ktorý možno nasadiť v súkromných cloudoch aj na AWS. Druhý návrh zdôrazňuje prenosnosť a flexibilitu cloudových vzorov, pretože ich možno prispôsobiť rôznym prostrediam a prípadom použitia.

Nakoniec sú vykonané experimenty na vyhodnotenie výkonnosti a efektívnosti cloudových vzorov použitých v demonštračnej aplikácii. Výsledky týchto experimentov potvrdzujú, že vybrané vzory skutočne poskytujú očakávané výsledky z hľadiska škálovateľnosti, odolnosti a výkonu.

Na záver táto práca predstavuje analýzu cloudových vzorov a ich použitie pri riešení bežných problémov v prostredí cloudu. Demonštračná aplikácia navrhnutá pomocou Kubernetes a nasadená na AWS slúži ako praktický príklad potenciálu cloudových vzorov, pričom vykonané experimenty potvrdzujú ich účinnosť. Tento výskum prispieva k lepšiemu pochopeniu cloud computingu a poskytuje cenné poznatky pre vývojárov a architektov pracujúcich s cloudovými aplikáciami.

Cloud Computing Application Design Patterns

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý, Ph.D. All the relevant information sources that were used for this thesis are properly cited and included in the references list.

. . . .
Matej Kolesár
May 16, 2023

Acknowledgements

I would like to thank my thesis supervisor, RNDr. Marek Rychlý, Ph.D. for his time and willingness to help when I was not sure how to proceed and for the patience that he had while discussing all the problems that were encountered.

Contents

List of Figures

Chapter 1

Introduction

Cloud computing is a field that is gaining more and more popularity in recent times. Unfortunately, the cloud environment is different from the usual environment that some programmers may be used to. This difference, if not handled properly, can result in the final application not being able to utilize all of the features and benefits that are available in the cloud environment. As such, identifying key areas and problems that can commonly occur in cloud environment is key to be able to properly design an application that can utilize the cloud efficiently. In this thesis, these correct approaches and the proper recommended solutions are going to be referred to as cloud computing patterns. Patterns will allow us to standardize certain situations and propose ways to handle them. The patterns can occur in many forms, from situations that are very specific and are optimized for certain data flow to more high level ideas that should be present through the application.

This thesis will aim to introduce the most common types of challenges that can occur, how and when they occur, and list available design patterns that handle these issues correctly. Chapter 2 is focused on a brief introduction of all types of available solutions that are based on the provided resources. Afterwards, the locations of these resources are compared and the benefits of each solution are listed. After this analysis, the current biggest cloud providers are listed with some of the services that they offer. At the end of the chapter, the most common technologies that are used and benefit the most from the cloud environment are mentioned. Chapter 3 first introduces the IDEAL properties that are expected in the cloud environment. With the core properties established, patterns that adhere to these properties are chosen. The focus is then to introduce the event driven architecture, n-tier architecture and microservices. As the chosen solution is the microservice architecture, a deeper investigation into the properties and behaviour of this solution is performed. Key areas that microservices need are identified and later used when creating the design. Chapter 4 introduces the components from which the chosen solution is created. These components are present in all types of architectures and some require a close to mandatory use in the cloud environment. The first analysed part is the workload type that the components can experience. Different types of workload are identified. These workloads are taken into account when creating the microservices in the design as the goal is to showcase all types of workloads. Next, the different types of storage solution are analysed and the one that fits the best is chosen for the design. Then, the database solutions are analysed as they are a core functionality in nearly every application. Afterwards, the communication interfaces are described and established. The chapter ends with a look into the management components that help to monitor the cloud, automate the solution and provide a secure network. At the end of the chapter, the structure of the diagrams is proposed as there is no current standard and each provider uses his own design. Chapter 5 goes into the actual designs that were created. The only difference is that one design is optimized for use in AWS where the message broker and the storage solutions is substituted with AWS solutions. The rest of the chapter goes into the challenges that the application faces and how they are handled. The behaviour is analyzed for each use case and the designed solution is explained. Chapter 6 goes into the implementation details and explains what kubernetes files are created and their purpose in the solution. After the kubernetes files are explained, another brief description is given to each microservice, where the behaviour of the exact microservice is defined and available communication interfaces are explored. Chapter 7 is focused on the deployment of the implemented application on AWS. The chapter also includes experiments, which are performed to test whether the patterns that were used in the design have the expected effect. Many of the patterns are based on the proper design of the application, with which many problems that could occur are handled preemptively. The rest of the design patterns are mostly component based and are separated into groups depending on what is the expected benefit from the pattern.

Chapter 2

Current state of cloud solution providers

Cloud computing has become an important part of the current in the current day and age, offering organizations the ability to leverage powerful computing resources on a flexible, on-demand basis. There are many cloud computing solutions available, each with their own unique features, advantages, and limitations. In this chapter, we will explore some of the most widely used cloud computing solutions, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) and Function as a Service (FaaS). We will examine the technical aspects of these solutions, including their architecture, deployment models, scalability, and security considerations. Additionally, we will explore the key providers in the cloud computing market, including Amazon Web Services, Microsoft Azure, Google Cloud Platform, and others. Currently, there are many approaches and computing models to choose from. All these solutions provide different benefits and contain different features. Depending on how we look at it, we can divide them between solutions based on infrastructure or solutions based on a concrete cloud provider.

2.1 Existing cloud computing solutions

In cloud computing, there are a few different models that are available for use. They are usually separated into four models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) and Function as a Service (FaaS). The comprehensive comparison can be seen in figure 2.1.

IaaS

Infrastructure as a service (IaaS) [35] is a cloud service which supplies the needed storage, networking servers, and virtualization resources. This form of service removes the need for an on-premise data center and the management and maintenance of physical servers is the responsibility of the IaaS provider. The client has more flexibility and can install what software they need, but are responsible for security updates on their software as no software, including OS, is provided by the IaaS provider. The IaaS provider is hosting these resources in either a public cloud, private cloud or hybrid cloud. One of the existing solutions is for example AWS EC2.

PaaS

Platform as a service (PaaS) [38] [12] is a cloud service which provides computing mechanisms for deploying applications, designing applications for the cloud, pushing applications to their deployment environment, using services, migrating databases or a build integration tool. PaaS have features like built farms, routing layers, or schedulers that dispatch workloads to different virtual machines. Many of the current solutions for PaaS have a container foundation for running platform tools(i.e. Openshift, Cloud Foundry, Tsuru). There are still differences between how these platforms handle certain problems. While Cloud Foundry handles stateful services by running them in VMs and stateless in containers, Openshift does not differentiate and runs both of them in containers. Some existing examples are AWS Elastic Beanstalk, Windows Azure and Openshift.

SaaS

Software as a service (SaaS) [15] [46] is a software distribution model in which a cloud provider provides software and makes it available to end users over the internet. A software provider will either host the application and related data using its own servers, databases, networking and computing resources, or it may be an independent software vendor (ISV) that contracts a cloud provider to host the application in the provider's data center. In this model, the provider is responsible for the setup, maintenance and support of the software and the client receives ready made solutions. In comparison to traditional software that runs on operating systems, SaaS is in most cases deployed on an existing PaaS system or a specialized SaaS infrastructure. Some of the existing solutions include BMC Software, CyberArk, AWS Cohesity.

FaaS

Function as a service (FaaS) [7], or more commonly known as serverless computing model, allows the cloud provider to flexibly control the distribution of computer resources. In this model, the management of resources and dynamical provisioning is the responsibility of the cloud provider. Therefore, the cloud client is relieved of the responsibility of managing, scaling, or provisioning any servers. Users upload their code, which is then executed by the cloud provider and scaled automatically in response to the volume of incoming requests. As the customer only pays for the precise resources and time that their code uses, this might result in cost savings. Some of the existing solutions are Amazon Lambda, Google functions, Azure Functions.

2.2 Cloud location

Cloud computing can also be looked from the point of deployment. The most used options are to either to use private cloud where the user has to deploy the application and handle all the management, utilize the options of vendors and to rent their infrastructure or, the most common case, a hybrid one where the user has some local private cloud but also needs to scale while using public cloud. Usually, private cloud consist of a data storage while the applications run on the public cloud.

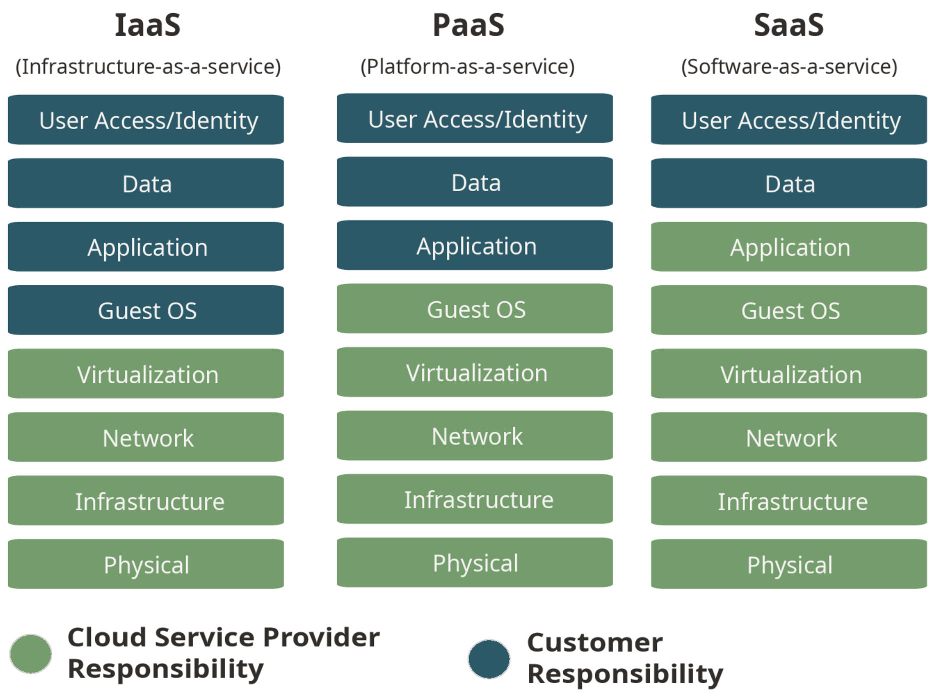


Figure 2.1: Comparison between SaaS, PaaS and Iaas.

Private cloud

Private cloud [37] is a solution for cloud computing which is not accessible to the public but only certain individuals and groups. It requires that companies invest into architecture, maintain this architecture and are responsible for all accompanying cost. They are also responsible for all security measures that are needed from physical security to network security. Usually, there is no sharing of infrastructure, no multi tenancy issues and zero latency for local applications and users which simplifies some problems faced in public clouds.

Public cloud

Public cloud [26] is defined as computing services offered by third-party providers over the public Internet, making them available to anyone who wants to use or purchase them. They may be free or sold on-demand, allowing customers to pay only per usage for the CPU cycles, storage, or bandwidth they consume. In this model, the cost of maintenance need for hardware and physical security is moved to the owner of the cloud. Possessing a large number of wide spread world resources enables providers to offer a consumer different choices to select appropriate resources while considering the quality of service (QoS).

Hybrid cloud

A hybrid cloud is one in which applications are running in a combination of different environments. Hybrid cloud computing approaches are widespread because almost no one today relies entirely on the public cloud. An example of this could be an on-premises data center, and a public cloud computing environment.

2.3 Concrete providers

After determining what type of service is the most useful one, there are many options. There are many providers that offer public cloud or can be used in a hybrid setup. These providers operate large data centers filled with computing resources such as servers, storage, and networking equipment, which they make available to customers.

AWS

Amazon Web Services (AWS) [1] [22] is one of the oldest cloud providers and provides a range of computing services like cloud storage, database service, analytics, network, mobile computing and enterprise services. AWS services are delivered to customers via a network of AWS server farms located throughout the world. Fees are based on a combination of usage (known as a „Pay-as-you-go“ model), hardware, operating system, software, or networking features chosen by the subscribe, required availability, redundancy, security, and service options. Currently, AWS has 30 availability regions in the world.

Azure

Azure [31] is Microsoft's cloud computing platform, and it offers a similar range of services to AWS, with a focus on cloud computing, storage, database, analytics, and artificial intelligence (AI). Azure has multiple capabilities such as software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS) and has the needed support for integration with other Microsoft services.

Google cloud

Google Cloud [2] is the newest of these platforms, and it offers a smaller range of services, with a focus on cloud computing, storage, and analytics, as well as a number of machine learning and AI services. Google cloud provides infrastructure as a service, platform as a service, and serverless computing environments. One of the downsides is that it lacks SMTP support and the default components that are necessary need around 12% of the available RAM (percentage depends on machine).

2.4 Container and orchestration solutions

Our main focus for deployment are going to be containers [41], specifically docker containers, which is one of the most wide spread deployments methods. As the name suggests, the main component in this approach is a container. A container is a standard unit of software that packages up the code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

Docker

Docker [41] is a set of platform as a service (PaaS) products that provides a facility to automate applications when they are deployed into containers. Docker adds an extra layer of deployment to the containerized applications to allow to execute the applications. It is designed to provide a lightweight environment in which code can be run efficiently and, moreover, it provides an extra facility of the proficient work process to take the code from the computer for testing before production.

Kubernetes

Kubernetes [9] is an open-source container orchestration system that allows users to manage and deploy large numbers of containers at scale. Kubernetes provides features such as automatic scaling, self-healing, and load balancing, which make it easier to deploy and manage applications in the cloud. This can be useful for applications that need to handle large volumes of traffic or data, or that require high availability and resilience. Kubernetes works directly with the container through Containerd, or replacing Docker with a runtime that is compliant with the Container Runtime Interface.

Openshift

OpenShift [40] is a service(PaaS) built around Kubernetes. It provides additional features and tools that make it easier to develop, deploy, and manage applications in the cloud. OpenShift also integrates with a number of other technologies, such as Jenkins and Git, which can be used to automate the application development and deployment process.

Chapter 3

Cloud architecture design pattern

One of the most important parts when creating an application for cloud environment is the chosen architecture. Certain architecture designs benefit greatly from the features that cloud environments provide. There are certain properties that are expected to fully utilize the advantages of the cloud. These properties are isolated state, distribution, elasticity, automated management, and loose coupling (IDEAL properties) [18]. Aside from these general properties another major part are the data types which the application will use. We will concentrate on the general properties as there are many types of data and it is not possible to analyze all of them in-depth.

3.1 IDEAL properties

Before looking at different architecture choices that are available, it is necessary to define properties that should be present in them to fully utilize the advantages of the cloud. The properties will be defined from a high level and should give a general view of what is expected in the architectures. IDEAL properties (isolated state, distribution, elasticity, automated management, and loose coupling) [18] can be applied to all levels when designing the architecture or component behaviour.

Isolated State

Isolated state is meant to separate the state information of a microservice from other microservices as much as possible. The state information represents the data state of the interaction with an application and the data handled by the application. Ideally, the data is handled through data stores. This isolation enables easier scaling and resiliency with respect to failures of the application.

Distribution

Distributing the application functionality among multiple components to be deployed on multiple cloud resources represents another essential step. Applications have to consider the distribution and the scaling support of their cloud environments to effectively utilize it. One of the options for applications is to rely on redundant components.

This can especially be the case if the cloud provider assures environment-based availability, which is the availability of the complete environment and not of single IT resources hosted in it.

In practise, the functionality of the application is divided into multiple independent components that provide a certain function. These components are separated and split up depending on the architecture into a tier based architecture 3.2, event based 3.3 or microservice architecture 3.4.

Elasticity

Elasticity allows the application to provision and decommission resources while the application is running. This action occurs without affecting the user and it is possible to either increase the available resources (vertical scaling) or increase the number of instances (horizontal scaling).

Automated Management

Automated management is needed to promptly react to changes within the application or environment and apply the needed changes(i.e. change resource allocation, restart service, etc.). For this, the management system needs to have the ability to see the state of each service and have a defined behaviour in case of component failure to know which operations to execute to fix the problem. This should ideally be done without human interaction.

Loose coupling

Loose coupling ensures us that calls between applications, scaling, deployment, failure handling or updating are handled independently. This is by keeping the dependencies between components to a minimum.

Information exchange between applications and their individual components as well as associated management tasks, such as scaling, failure handling, or update management can be simplified significantly if application components can be treated individually and the dependencies among them are kept to a minimum.

3.2 N-tier Cloud Application

In an N-tier architecture [5] [42] the application is separated into multiple tiers which consist of similar logic. These application can have a closed architecture where a tier can only call the next tier which is under it or an open tier architecture where a tier can call any of the tiers below it.

Depending on the degree of separation there are the following designs:

1. 2 tier architecture where the data access is separated from the rest of the application
2. 3 tier architecture which has a presentation tier, business tier and data access tier
3. 4 tier architecture which has a presentation tier, data service tier, business tier and data access tier

There are more tiers available but they are not usually used and are only useful in special circumstances.

Two-tier applications provide the minimal separation from data tier. Their implementation is in most cases easier at the cost flexibility and the adaptation or change of features.

A three-tier application compared to a two-tier separates the presentation tier and business tier. The functionality of these tiers is as follows:

The Presentation tier is the topmost tier of an application. This is the tier seen when using the software(Interface, web pages). By using the software we access web pages. Its main function is to communicate with the application tier. This tier passes information which is given by the user in terms of keyboard actions and mouse clicks to the application tier.

The Application tier is also known as the Business logic tier. In this tier, we can find logic controls and functionality that processes data received from the presentation tier and database tier. It acts as an intermediary between the presentation and database tier.

The Database tier is the tier that stores data in the storage. It contains methods that connect to the database and performs the required actions needed. These are Insert, update or delete. The three tier architecture can be seen in the following figure 3.1.

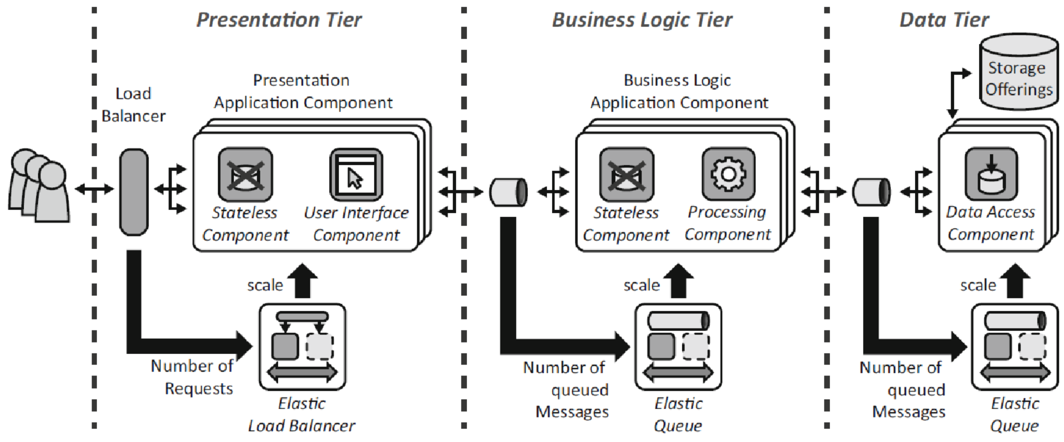


Figure 3.1: Example of three tier application in cloud environments [18]

3.3 Event-driven architecture

An event-driven architecture [33] consists of event producers that generate a stream of events, and event consumers that listen for the events.

In most cases, events are created immediately after an action occurs, so consumers can respond to events right after they are created. Consumers and producers are decoupled. A producer does not know which consumer is listening to the events and a customer does not know other consumers or the producer of the event. There are many types of consumers. These include Competing Consumers pattern, where consumers pull messages from a queue and a message is processed just once if note error occurs, Selective Consumer or Event-Driven Consumer.

An event driver architecture can also work with the publish/subscribe (pub/sub) model or event streaming model. In the pub/sub model, the messaging infrastructure keeps track of subscriptions. When an event is published, it sends the event to each subscriber. Events are not stored and only sent to current subscribers, which means that new subscribers can not receive or see already published events. In the case of event streaming, events are written to a log. Events are strictly ordered and durable. A client can read from any part of the stream and is responsible for advancing its position in the stream. This allows client to see historic events.

These types architecture is ideally used to process real-time information with minimum time lag, complex event processing, aggregation over time window and high volume and high velocity of data.

3.4 Microservice

A microservices architecture [45] [16] consists of a collection of small, independent applications. Each microservice can be deployed independently, scaled independently and has a single responsibility. Single responsibility is meant from a functional view, for example, reading a file, validating content and providing little modification before sending it to other microservices. By breaking down applications into smaller, self-contained services, microservices enable developers to work on individual services independently, without impacting other parts of the application. With this approach, it is possible to use multiple different technologies as long as they are compatible with their respective interfaces.

Microservices have many unique characteristics compared to other traditional architectures like SOA. The attributes that usually different from other existing approaches are size, bounded context and independency.

The size is should be smaller when compared to typical services. They should be small enough to be developed, tested, and deployed independently, while also being large enough to provide meaningful business value. Ideally, a microservice should be able to be developed and deployed by a small team, with minimal dependencies on other services. The size of a microservice can vary and the size difference can be fairly significant. However, it should be designed to be easily maintainable and extendable, with a clear separation of functionality. The benefits of smaller microservices are improved agility, faster development cycles, and easier deployment, as well as easier scaling and better fault tolerance.

Bounded context is a key concept in Domain-Driven Design (DDD) that refers to the idea of defining explicit boundaries around a particular domain or subdomain of an application. The idea is to create a clear separation of concerns between different parts of the application, and to define explicit boundaries and interfaces between them. A bounded context defines the scope of a microservice, which represents a specific business capability or functionality. Each microservice should have a clear and well-defined bounded context, which helps to ensure that it has a clearly defined purpose, and that it has a clear understanding of the data and behavior that it needs to manage. By defining bounded contexts, developers can design microservices that are highly cohesive and loosely coupled, which helps to minimize dependencies between services and makes it easier to manage and evolve the application over time. This approach also enables teams to work more independently and with greater autonomy, since each team can focus on a specific bounded context without needing to coordinate with other teams.

Independency, which refers to the ability of each microservice to function independently with minimal dependencies on other services or components, represents another important concept. This means that each microservice should have a clear and well-defined purpose, and should be able to operate autonomously, without needing to rely on other services or components for its functionality. This does not mean that the microservices can not have any dependencies but the communication to other services should be through well-defined interfaces and ideally loosely coupled.

After establishing the main differences, the key characteristics can be summarized into the following:

1. Flexibility – A system is able to keep up with the ever-changing business environment and is able to support all modifications that are necessary for an organisation to stay competitive on the market
2. Modularity – A system is composed of isolated components where each component contributes to the overall system behaviour rather than having a single component that offers full functionality
3. Evolution – A system should stay maintainable while constantly evolving and adding new features

3.4.1 Impact on quality and management

To better grasp the impact that microservices have, it is necessary to look at the quality of the underlying system.

The availability and reliability of microservices [45] is a major concern since they are distributed across different systems. Availability can be impacted by network failures, hardware failures, or software bugs. However, microservices offer several advantages that can improve availability. By breaking down the system into smaller services, each service can be scaled independently, allowing for better resource allocation and fault isolation. Additionally, since services are loosely coupled, failures in one service will not affect the availability of other services

Since microservices are designed to be loosely coupled, this results in the fact that the services are designed to have minimal dependencies on one another. This feature enhances the maintainability of a system by reducing the cost of modifying services, fixing errors, or adding new functionality. However, it is still possible to compromise maintainability by writing complex and messy code. To address this issue, the „you build it, you run it“ principle has emerged as an additional means of promoting maintainability in microservices. This principle emphasizes the responsibility of individual teams to build and run their services, which fosters a deeper understanding of each service's business capabilities and roles. By increasing visibility and accountability, this principle can help to ensure that the services remain maintainable and scalable over time.

The performance of microservices can vary depending on several factors, including the specific implementation, the number of services, and the communication protocols used. While in-memory calls are much faster than sending messages over the network, the prominent factor that negatively impacts performance in the microservices architecture is communication over a network. This is due to the network latency being much greater than that of memory. In addition, restrictions on the size of microservices indirectly contribute to the performance degradation by increasing the ratio of messages sent over the network compared to in-memory calls. Therefore, proper planning and API design are crucial to minimizing the amount of communication over the network and achieving optimal performance. Systems with well-bounded contexts and fewer connections between services will experience less degradation due to looser coupling and fewer messages sent. When designed and implemented correctly, microservices can offer superior performance compared to monolithic architectures by allowing services to be scaled independently.

Due to the independent nature of each component in a microservices architecture, it becomes possible to test each component separately, which greatly enhances the testability of individual components as compared to a monolithic architecture. Additionally, this isolation allows for a more focused testing scope, which can be adjusted based on the size of changes made. With microservices, it is possible to isolate parts of the system that have undergone changes and test them independently from the rest of the system. However, integration testing can become challenging, especially in a large system with multiple connections between components. Although each service can be tested separately, interactions between multiple services can lead to anomalies.

Chapter 4

Cloud component patterns

In this chapter we will look at what type of situations can be encountered in the cloud and how they can be solved. These are mostly very specific problems that are handled with special components. From problems with balancing workloads, to different option of storage solutions, different database solutions, multiple choice for forms of communications and finally different ways of managing and monitoring all these solutions.

4.1 Cloud Workloads

One of the major incentives for moving to cloud based solutions is the option to manage workloads. Workload management is a major part as it influences the scaling of the application. According to the workloads, the resources are provisioned or decommissioned to the needs of the application. This is done to ensure the increase capacity (concurrent users) and reliability of applications. It also used to save cost on resource in case of lower workloads during non-peak hours.

Static workload

Static workload [18] is a type of workload that is characterised by a nearly constant demands over a period of time. As such, there is no need to adjust the required resources and the resources are acquired in advance.

In ideal scenarios static workload is much better than elastic workload as there are the following benefits:

- Predictability – static workloads are more predictable than dynamic workloads, which makes it easier to plan and allocate resources to handle them. This can help to improve the efficiency and reliability of the cloud computing environment.
- Cost-effectiveness – static workloads do not require frequent changes to resource allocation, they can often be handled with a fixed set of resources. This can help to reduce costs and improve the overall cost-effectiveness of the cloud computing environment.
- Reliability – static workloads are more predictable, so they can be more easily managed and maintained. This can help to improve the overall reliability and uptime of the cloud computing environment.

Elastic workload

Elastic workloads [24] [32] are the most common type of workloads and they are tasks or applications that are expected to change frequently. To handle this type of workload reliably, the ability to dynamically and automatically scale up or down the amount of computing resources used based on the current and anticipated demand is needed. This allows applications to quickly and easily adjust their resources to meet changing demands. By their nature, elastic workloads are harder to manage than static workloads. The main advantage of elastic workload is that not many applications can depend on static workload in the real world. As such, being able to dynamically adjust the resources can decrease the energy consumption and be more cost effective while being able to provide the quality similar to application that expect static workloads. The workload will be divided into 2 main types. Workloads that are predictable are usually about events that are known but it is unsure how many resources the application requires. The other type is unpredictable workloads, where the need to scale up and down dynamically is needed.

Predictable workloads

Predictable workload changes [18] are periodic workload and are a very common in the real world (i.e. weekly status reports, rush hour, periodically scheduled task, etc..). These types of periodic tasks and routines occur in nearly all system and having a model allowing the decommissioning of resources during non-peak times can increase the cost effectiveness of the application. A special type of periodic workflow are once in a life time workloads that have peaks in big time intervals and are usually correlated to certain events or tasks. Continuously changing workloads [34] are usual in applications that experience steady continuous growth or decline. These changes are usually based on historic data or knowledge about the application and as such provision or decommission resources is done with the same rate as the workload changes are expected.

Unpredictable Workload changes

Unpredictable workloads [18] are based on unexpected increases or decreases in traffic to the application. Unplanned provisioning and decommissioning of IT resources is required to provide an uninterrupted and reliable service. The necessary provisioning and decommissioning of resources is, therefore, automated to align the resource numbers to the changing workload. One main factor for unpredictable is the time needed for resource provisioning. Provisioning of resources may take up to several minutes as such unpredictable changes may have certain bottlenecks till the resources are available. The same applies to decommissioning resources when they are no longer needed.

Load balancer

Load balancers [20] are ideal components to deal with problems regarding workloads. Formally defined load balancing is the process of distributing a set of tasks over a set of resources, with the aim of making their overall processing more efficient. Load balancing can optimize the response time and avoid unevenly overloading some compute nodes while other compute nodes are left idle. There are many existing algorithm for load balancing and they require metrics so they can be applied correctly. The metrics will be divided into the following ones:

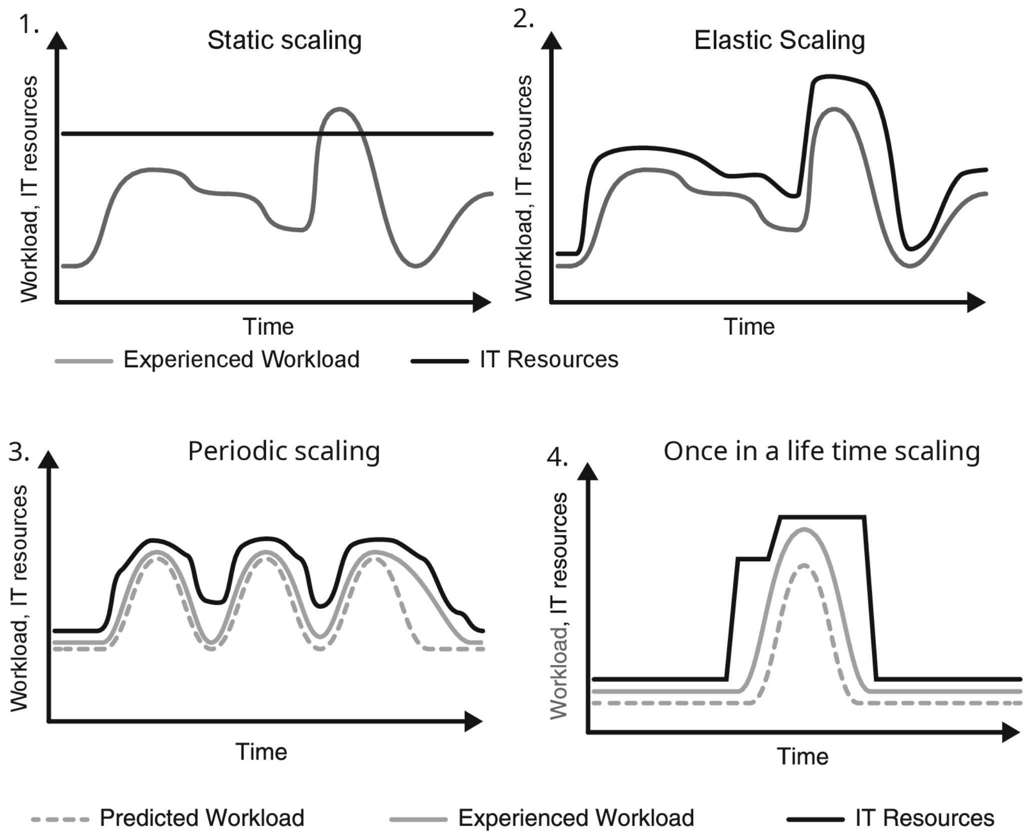


Figure 4.1: Workload used for traffic in static and elastic scaling [18]

- Throughput – this metric is used to calculate the number of processes completed per unit time.
- Response time – it measures the total time that the system takes to serve a submitted task.
- Makespan – it is used to calculate the maximum completion time or the time when the resources are allocated to a user.
- Scalability – it is the ability of an algorithm to perform uniform load balancing in the system according to the requirements upon increasing the number of nodes.
- Fault tolerance – it determines the capability of the algorithm to perform load balancing in the event of some failures in some nodes.
- Migration time – the amount of time required to transfer a task from an overloaded node to an under-loaded one

- Energy consumption – it calculates the amount of energy consumed by all nodes. Load balancing helps to avoid overheating and therefore reducing energy usage by balancing the load across all the nodes.

There are many options for existing solutions on how to handle load balancing. There is going to be a deeper investigation into the following one:

- Natural Phenomena-Based Load Balancing [20] – several load balancing strategies that are inspired by natural phenomena or biological behavior, for example, Ant-Colony, Honey-Bee, and Genetic algorithms.
- Agent-Based Load Balancing agent [20] – the dynamic nature of cloud computing is suitable for agent-based techniques. An agent is a piece of software that functions automatically and continuously decides for itself and figures out what needs to be done to satisfy its design objectives. A multi-agent system comprises a number of agents, which interact with each other. To be successful, the agents have to able to cooperate, coordinate and negotiate with each other. Cooperation is the process of working together, coordination is the process of reaching a state in which their actions are well suited, and in negotiation process, some parameters are agreed.
- General load balancing technique [20] – these load balancers include algorithms for First-In-First-Out (FIFO), Min-Min, Max-Min, Throttled, and Equally Spread Current Execution Load.

In the cloud environment, the load balancers may have many different responsibilities. Some may just distribute the load while others can also change the amount of provisioned resources. The load balancers that can also change the resources provisioned will be refered to as elastic load balancers. An elastic load balancer [27] is a management component that is provided with information from a load balancer that spreads out synchronous requests from human users or other application components among multiple component instances. Based on the number of distributed requests and possibly other utilization information, the required number of required component instances is determined. When determined, the necessary provisioning or decommissioning operations to reflect this number in the application are executed. The elastic load balancer invokes these operations provided by the interface of the elastic infrastructure or the elastic platform. A practical implementation of this is in Kubernetes the HorizontalPodAutoscaler (HPA), which handles the scaling of the pods.

4.2 Data storage

Data storage is one of the more popular usages of cloud environments. The incentive to move data to cloud storage is first from an availability standpoint, where the data can be accessed as long as there is a working connection to the cloud storage. Another incentive is if public cloud providers are used with their existing solutions, then there is no need to invest into on-premise architecture, which is not only a cost saving but also simplifies the maintenance of the storage and possible future expansion if needed. The main advantages that the cloud storage provides is a large pool of storage, with three significant attributes: access via Web services APIs on a non persistent network connection, immediate availability of very large quantities of storage, and pay for what you use. It supports rapid scalability.

The following sections will dive into the different storage types and when they should be used.

File Storage

File storage [18] is a hierarchical storage system in the cloud that provides shared access to files. Many applications need to access shared files and require a file system. This type of storage is often supported with a network-attached storage (NAS) server. Some of the available solution by vendors are Amazon EFS, Azure Files.

Block Storage

Block storage [29] [30] offers virtual hard drives as IaaS often used for virtual servers offered in an elastic infrastructure. Storage in such services is organised as blocks. This emulates the type of behaviour seen in traditional disks or tape storage through storage virtualization. Blocks are identified by an arbitrary and assigned identifier by which they may be stored and retrieved, but this has no obvious meaning in terms of files or documents. One of the few options to differentiate files is to separate the block storage into volumes. In cloud, there is usually a block storage system which serves as a middleware layer that accesses the stored files. These systems usually handle load balancing, cache efficiency, and storage cluster management. Some of the existing solutions by vendors are Amazon Elastic Block Store, Azure Disk Storage, Persistent Disk.

Instance stores are another form of cloud-hosted block-level storage. They are best used for temporary storage such as caching or temporary files, with persistent storage held on a different type of server.

Object Storage

Object storage [6] is a data storage architecture that can store large unstructured data. This includes videos, photos, web pages, audio files, sensor data, and other types of web content. Objects are discrete units of data that are stored in a structurally flat data environment. There are no folders, directories, or complex hierarchies as in a file-based system. Every objects consists of data, metadata that describes the object and a unique ID that is used to locate the object. This flat architecture removes some issue of complexity and scalability that exists in standard hierarchical file systems. Current available solution for object stores include Amazon S3, Google Cloud Store(GCS), Swift and MinIO.

4.3 Database options

Database is an organized collection of data. This data is either structured and then it is most likely better to use a relational database, or it is semi-structured/unstructured where the NoSQL databases are more likely to be used. While the standard SQL types are more rigid due to the ACID principle and their requirement for consistency, NoSQL databases are ideal for cloud environments as they already have characteristics that are similar. First, we will look at the CAP theorem that describes the ways we can handle distributed data. This is important for NoSQL as they can be eventually consistent.

The CAP theorem [8] [36] states that any distributed data store can provide only two of the following three guarantees:

1. Consistency – every read receives the most recent write or an error.
2. Availability – every request receives a (non-error) response, without the guarantee that it contains the most recent write.
3. Partition tolerance – the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

When a network partition failure happens, it must be decided whether to cancel the operation and thus decrease the availability but ensure consistency or proceed with the operation and thus provide availability but risk inconsistency.

NoSQL

A NoSQL (not only SQL) [21] database is used to store and retrieve data in other format than traditional relational databases. Nowadays NoSQL is used as an umbrella term for unstructured or semi-structured databases. As such we usually look for certain characteristics that they need to fulfill. These characteristics are simple and flexible non-relational data models, ability to scale horizontally, provide high availability, usually they do not support ACID principles and are BASE systems (Basically Available, Soft state, Eventually consistent) [11]. These characteristics make NoSQL data stores especially suitable for use in cloud environments.

Key-value database [25] is a type of NoSQL database that is commonly used in cloud environments. In a key-value store, data is organized and accessed using unique keys, and each key is associated with a corresponding value. Key-value storage is simple and efficient, and it is often used for high-speed, high-volume data access and retrieval. Document databases (Document Oriented Database) are a type of NoSQL database where the underlying storage structure used is a 'document'. Each Document Store differs in its implementation of data, however every solution assumes that data is enclosed and encoded in some standard format which may be XML, BSON, PDF, JSON or other standard formats. Each document is represented by a unique key which is a string (URI or path). An API or a query language is provided for fast retrieval of documents on the basis of its content (e.g. a query that retrieves all the documents in which certain field is set to some particular value).

Other types of NoSQL databases include Graph Database, Column Oriented Databases, Multidimensional Databases but we will not analyze all of them.

Some of the key advantages and features of NoSQL databases in the cloud are:

- High performance – NoSQL databases are designed for fast, efficient data access and retrieval. This makes them ideal for applications that need to process large amounts of data quickly, such as real-time analytics, search, and caching.
- Simple and flexible data model – most NoSQL databases use a model, which makes it easy to store, manage, and access data. This allows users to easily add, update, and retrieve data using unique keys, without the need for complex schemas or data structures.
- Scalability – NoSQL databases can be easily scaled up and down as needed, making it suitable for applications that require high availability and scalability. This can be useful in cloud environments, where resources are often shared and allocated on demand.

- Fault tolerance – NoSQL databases are often designed to be fault-tolerant, with built-in mechanisms for data replication, backup, and recovery. This can help protect against data loss and downtime in the event of hardware or software failures.

Relational Database

In relational database [25] [23] [13], data elements are stored in tables where each column represents an attribute of a data element with a well-defined semantic. These attributes may be used in data queries to make them more expressive. Furthermore,table columns may have dependencies in the way that entries in one table column must also be present in a corresponding column of a different table. These dependencies are enforced during all data manipulations. Another core functionality is partitioning to allow better scalability. Database partitioning serves for two purposes: to scale a single database to multiple nodes, useful when the load exceeds the capacity of a single machine, and to enable more granular placement and load balance on the back-end machines compared to placing entire databases.

The database can be realised directly on top of an IaaS cloud and are provided by the cloud provider(Amazon EC2). It can also be realised in the PaaS cloud(Amazon Relational Database Service, Microsoft SQL Azure). Another option is to use it as SaaS (PostgreSQL). While IaaS allows to better configure more details, SaaS makes it easier to use the database out of the box, but can have problems with configurability.

4.4 Cloud communication

This chapter will focus on the most common communication types that are used between applications. The most common one is the REST interface which can provide us with synchronous or asynchronous communication. The main focus will be on synchronous REST, as for asynchronous, there are other options available for asynchronous communication type. The difference between synchronous and asynchronous communication is not as important to showcase REST. A good solution to showcase asynchronous implementation are Message-oriented Middleware (MOM) which are based on a message provider and a message consumer.

REST

REST [19] [28] is an architectural style for designing distributed, scalable systems. REST is often used in cloud computing to provide APIs, or application programming interfaces, that allow applications and services to communicate and exchange data over the internet. Despite that, there are some disadvantages that the REST design has that are more visible in cloud environments. They can sometimes be inflexible and fragile. That is most seen in cases where there are needed modifications to already implemented use cases. This can make it less adaptable and scalable compared to other approaches in some cases.

Message-oriented middleware

Message-oriented middleware [17] [14] (MOM) is a type of software that is used to facilitate communication and data exchange between different applications and services in a cloud environment. MOM allows applications and services to send and receive messages asynchronously, without the need for direct connections or interactions between them. This makes it easier to build complex, distributed systems that can scale and adapt to changing workloads and requirements.

One common use case for MOM in the cloud is to decouple different components of a system. For example, a cloud-based e-commerce platform might use MOM to separate the front-end user interface from the back-end services that handle inventory management, payment processing, and order fulfillment. This allows each component to operate independently and asynchronously, which makes the system more resilient and scalable.

Another use case for MOM in the cloud is to enable real-time data processing and event-driven architectures. For example, a cloud-based social media platform might use MOM to process and analyze incoming data from user posts and interactions in real time. This allows the platform to generate insights, make recommendations, and take actions based on the data, without the need for batch processing or long delays.

In addition to these use cases, MOM can also be used in the cloud to support messaging patterns such as publish/subscribe, request/response, and point-to-point. This allows applications and services to send and receive messages using different communication styles and protocols, which can be useful in a variety of scenarios. There are also many possibilities on how to consume messages [18].

1. Exactly-once delivery – in this delivery type, there are no duplicate messages. This is because the consumers are not idempotent. To solve this upon creation of messages, each message is associated with a unique message identifier. This identifier is used to filter message duplicates during their traversal from sender to receiver.

2. At-least-once delivery – in this delivery type, there is no need to care about the duplication of messages and therefore it shall still be ensured that messages are received.

3. Timeout-based delivery – in this delivery type, the message is not removed from the queue right after the client receives it. If a message is properly received, it is not deleted immediately after it has been read by a client, but is only marked as being invisible. In this state, a message may not be read by another client. After a client has successfully processed the message, it sends an acknowledgement to the message queue and upon reception, the message is deleted.

4. Transaction-based delivery – in this delivery format the queue participates in a transaction. All operations involved in the reception of a message are, therefore, performed under one transactional context guaranteeing ACID behavior.

Depending on the solution that is used, different terminology can be used. The focus will be mainly on RabbitMQ [3] as it is the message broker that is used in the demo application. The terminology that is recommended by the developers of RabbitMQ will be used and explained. In this context, producing means nothing more than sending. A program that sends messages is a producer. A queue is the name for the post box where the data is sent. Although messages flow through applications, they can only be stored inside a queue. A queue is only bound by the host's memory and disk limits and it is essentially a large message buffer. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue. Consuming has a similar meaning to receiving. A consumer is a program that mostly waits to receive messages. Note that the producer, consumer, and broker do not have to reside on the same host. This is the case in most applications. An application can be both a producer and consumer.

From the design of the message-oriented middleware, the following features that are useful for cloud environment are expected:

- Asynchronous – allows applications and services to send and receive messages asynchronously, without the need for direct connections or interactions between them. This makes it easier to build complex, distributed systems that can scale and adapt to changing workloads and requirements.
- Decoupling – can be used to decouple different components of a system, allowing them to operate independently and asynchronously. This makes the system more resilient and scalable, and it allows each component to evolve and be updated without affecting the others.
- Real-time processing – enables real-time data processing and event-driven architectures, allowing systems to generate insights, make recommendations, and take actions based on incoming data in real time. This can be useful in a variety of scenarios, such as social media, e-commerce, and IoT.
- Flexibility – allows applications and services to use different messaging patterns and protocols, such as publish/subscribe, request/response, and point-to-point. This makes the system more flexible and adaptable, and it allows different components to communicate in the way that is most appropriate for each scenario.
- Scalability – support large volumes of messages and high concurrency, making it easier to build systems that can scale up and down as needed. This can be useful in cloud environments, where resources are often shared and allocated on demand.

The different models of delivery and the inner workings from a high level point of view can be seen in figure 4.2.

4.5 Multitenancy patterns

In cloud computing, multitenancy [44] [12] means that multiple customers of a cloud vendor are using the same computing resources. Despite the fact that they share resources, cloud customers are not aware of each other, and their data is kept totally separate. Multitenancy is a crucial component of cloud computing as the only proper way to not allow any way for multitenancy to exist would be resource sharing. Data multi-tenancy is the most explored approach under multitenancy, and is often implemented on top of a database. There are three main approaches for data management in a multitenant deployment: separate databases, shared database with separate schemas and shared database with shared schemas. Aside from these, there can also be multi tenant behaviour on middleware. This comes comes with a big security issue as is described in [4] where this approach leads to many security issues identified by security experts. Altogether, there are three types of components that can exist in multi tenant environment as defined in [18]:

- Dedicated components – some also called single tenant because in this approach, components are specifically developed to be used by different tenants. They ensure isolation between tenants by controlling tenant access, processing performance used, and separation of stored data. It also allows for the most customizability options as are no dependencies on the behaviour of other tenants.

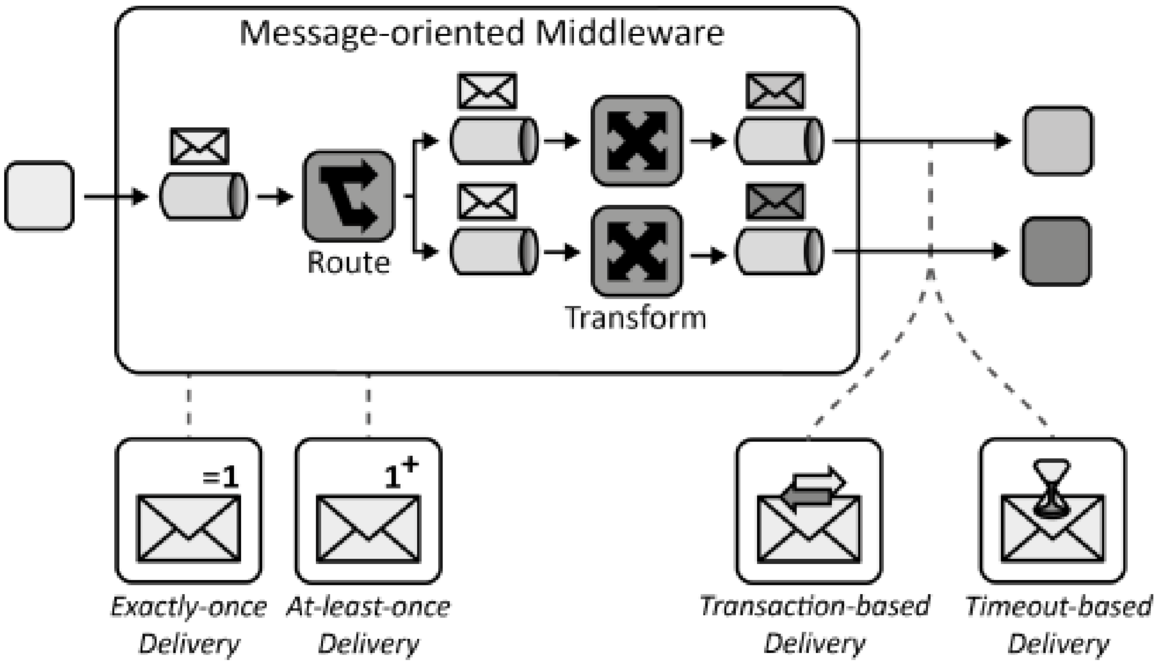


Figure 4.2: Message oriented middle with standard components and message processing types. [18]

- Tenant-isolated – this type of component allows to share resources of components while completely separating the logic of tenants. This causes duplicity during the implementation and harder maintenance but allows resource sharing.
- Shared Component – these components are used by all tenants but the logic of the component handles tenant specific behaviour. This model is most prone to security issues but also provides very good resource management.

4.6 Cloud management

The following section will focus on patterns that are used in management to deploy, monitor and manage deployed solutions in a cloud environment [39]. Patterns of this category describe how management functionality can be integrated with components providing application functionality. We will operate on the assumption that there exists a container management systems and analyse the characteristics of such system. First, we need to define the problem that these systems address. As containerisation had a big increase in recent types, it did come with an increase in complexity. With different types of architectures and solutions, there came a need to properly deploy, manage and monitor these resources. We will analyze the problems one by one and provide solutions to them.

Resource utilization and performance

Resource profiling [47] is needed to monitor the state of the cloud. To be effective, resource management needs to achieve an appropriate balance between adjusting resource allocations in response to detected changes in demand, and adjusting resource allocations in response to predicted demand, with predictions typically being based on historical measurements.

This is done by first collecting metrics from the environment (i.e. CPU usage, memory usage, etc..) and their analysis. Afterwards, there are defined procedures on how to handle these situations and whether to provision or decommission resources. This management of resources allows us to better utilize resource for application that needs them, which in returns increases performance. The solution for this is to have a standardized monitoring system which monitors whether a container is running or not. Aside from that, there is a need to monitor the resource usage for each component. This is in most cases not a problem and each container has its CPU and memory statistics.

Automation and orchestration

Container orchestration [10] is a method that allows cloud and application providers to specify how to choose, deploy, monitor, and dynamically manage the configuration of multi-container packaged applications in the cloud. During runtime, container orchestration handles the deployment, execution, and maintenance phases. Typically, container orchestrators provide several key features, including resource limit control, scheduling, load balancing, health check, fault tolerance, and autoscaling. By using resource limit control, providers can reserve a specific amount of CPU and memory for each container, which can be used to make scheduling decisions and prevent interference among containers. Container managers provide APIs that limit the amount of memory and CPU used and the specific CPU, which leverages the resource limit control features. However, while a container can consume all available resources on the underlying system, container managers set restrictions on resource utilization. There are many actions the orchestrations can do.

The scheduling feature in container orchestration determines how many containers should be placed on particular nodes based on resource constraints or node affinity. More advanced schedulers can be integrated as external components. Load balancing is responsible for distributing the workload among multiple container instances. The default policy is usually round-robin, but more complex policies can be implemented using external load balancers. Health checks are used to ensure that a container is able to handle requests, and they typically involve checking TCP/UDP/SSH ports for connectivity and HTTP requests for proper response.

Fault tolerance in container orchestration can be achieved through two main methods: replica control and high availability control.

- Replica control involves maintaining a specified number of container replicas. Health checks are used to identify any faulty containers, which are then destroyed and replaced with new instances to maintain the desired number of replicas.

- High availability control refers to the ability of a system to continue operating even in the event of hardware or software failures. This can be achieved through techniques such as load balancing, automatic failover, and data replication.

Autoscaling allows to automatically add and remove containers. The implemented policies are threshold based (on CPU and memory utilization) but in some cases it is possible to plug-in more sophisticated autoscaler or to define custom autoscaling policies.

Availability controllers provide the capability to set up several orchestration managers to have continuous control over the application, in the event of a failure or overload of an orchestrator node. The same method used to create high availability controllers can also be utilized for building scalable controllers.

Security

Cloud security [43] is a shared responsibility with your cloud service provider. Cloud management plays a very active role in protecting your data, applications, and services in cloud environments. To establish a secure application, a proper authentication method must be provided. This could be achieved by having an authorization server with which all components communicate or is used for at least the login validation. Leading cloud management tools offer machine-learning capabilities for robust threat intelligence and detection and help streamline security monitoring and processes. There are many security issues that come with cloud solutions.

One of them is the separation of data storage. While it is more cost effective to not have on-premise infrastructure, the removal also means a lack of control and inability to verify the security. This requires the customer to have full trust that the cloud provider is providing measures to ensure the safety of the data physically and also prevents data leaks. There are many more security issues as can be seen in table 4.1.

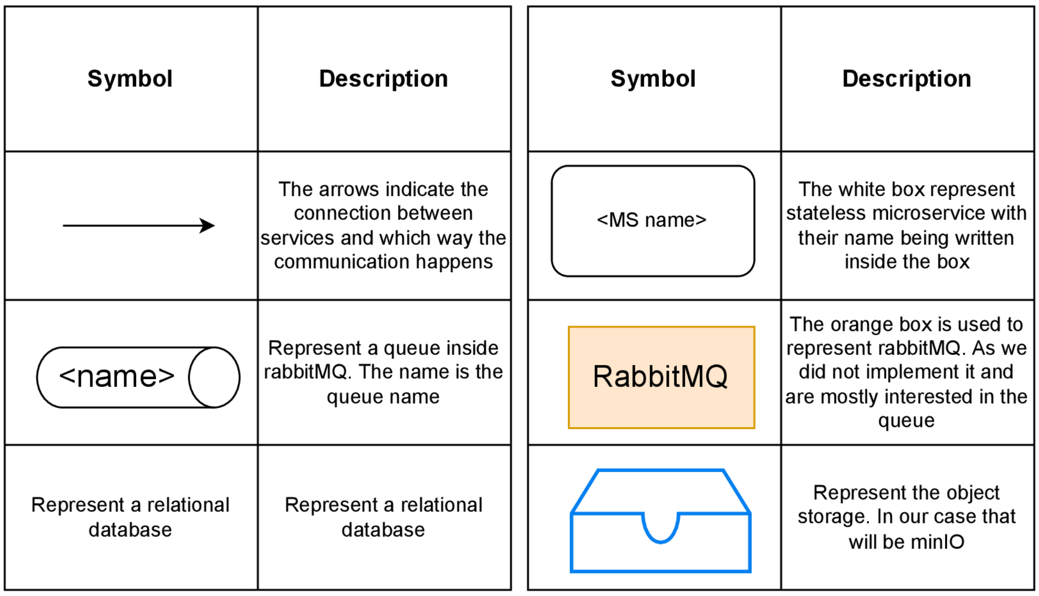
Security Topic	Security issues
VMs image management	Cryptographic overhead due to large size images VMs theft and malicious code injection Overlooked image repositor
Virtual machine monitor	Hypervisor failure, single point of failure Un-trusted VMM components Transparency of VMM Lack of monitor GUI VMM separation, inspection, and interposition
Network virtualization	Twofold traffic, limited network access, inapplicability of standard security mechanisms
Mobility	Packet sniffing and spoofing VM cloning VM mobility Generation of untruth configurations
Issues in virtual machine	Side-channel attack VM data extraction Filtration attack Memory deduplication
Malware	Spreading of malware onto VMs Metamorphic engines

Table 4.1: List of possible of the vulnerabilities in cloud. [43]

4.7 Diagrams

There is no universally accepted standard for representing microservices in diagrams, and different organizations may use different approaches depending on their specific needs and preferences. One standard that is used is based on the diagrams from AWS and one custom box like diagram where the microservices will be represented with boxes which contain their names.

Additionally, microservices architecture is highly modular and dynamic, with services being added, removed, and modified frequently. This means that any diagram representing a microservices architecture must be able to adapt to changes in the system over time. The definitions of the custom diagrams are in the following figure 4.3. For the AWS diagram, there is an official documentation where the components are described.



Symbol	Description	Symbol	Description
→	The arrows indicate the connection between services and which way the communication happens	<MS name>	The white box represent stateless microservice with their name being written inside the box
<name>	Represent a queue inside rabbitMQ. The name is the queue name	RabbitMQ	The orange box is used to represent rabbitMQ. As we did not implement it and are mostly interested in the queue
Represent a relational database	Represent a relational database		Represent the object storage. In our case that will be minIO

Figure 4.3: Legend of diagram components that are used in design.

Chapter 5

Demo application Design

The focus of this chapter will be on design of an application taking into account the patterns that were mentioned in the previous chapters 3 and 4. In order to do that, it is first needed to establish what our demo application should do and what behaviour do we expect. The application will be used by different users to save files to the cloud. The users will be able to save files, delete files or share their existing files with other users. When they share their file, the user that they shared it with will be notified about it. Users will be periodically billed for the provided service.

This is the main functionality that the application should provide. Some of the implementation will not be done(e.g. actual email sending for notification or billing) as it does not necessarily add anything to the verification of the patterns. With that in mind, there are 2 designs that are created. The first design does not use any of the services provided by the provider. By services are meant FaaS 2.1 or SaaS 2.1. Only IaaS 2.1 or PaaS 2.1 will be used. This can be seen in the figure 5.1.

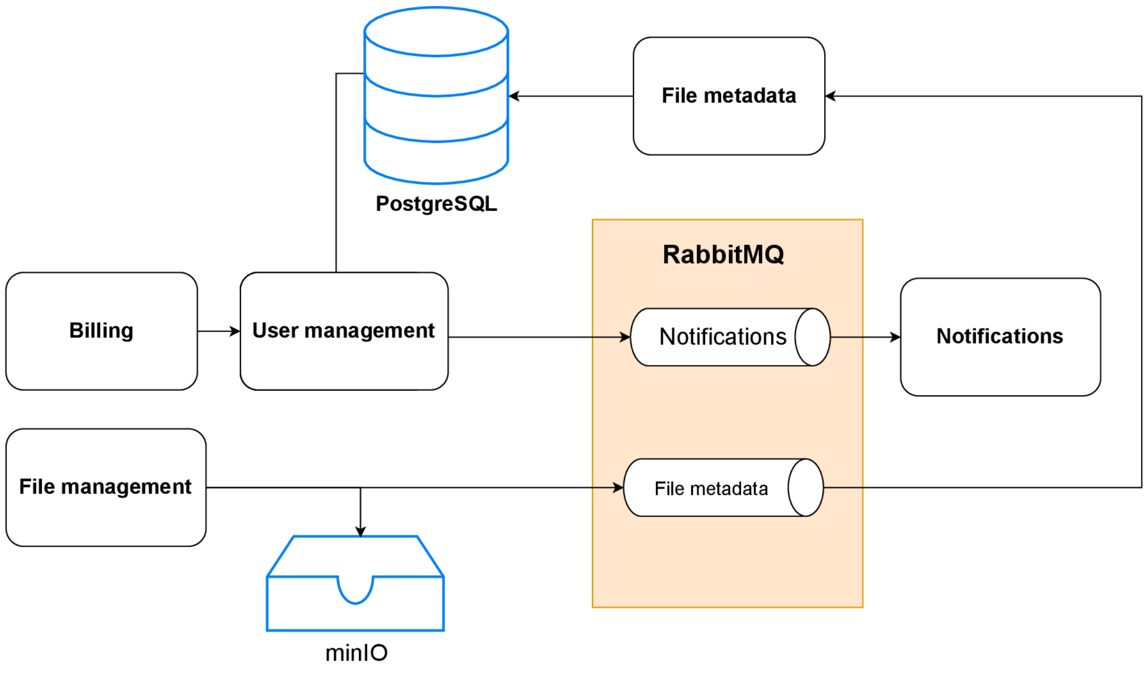


Figure 5.1: Design of demo application

The second design will use some of the services that AWS provides. Mainly, the storage solution will be swapped to S3 and the message broker to AmazonMQ. This is done to see the impact of removing these services from the node. The design can be seen in the figure 5.2.

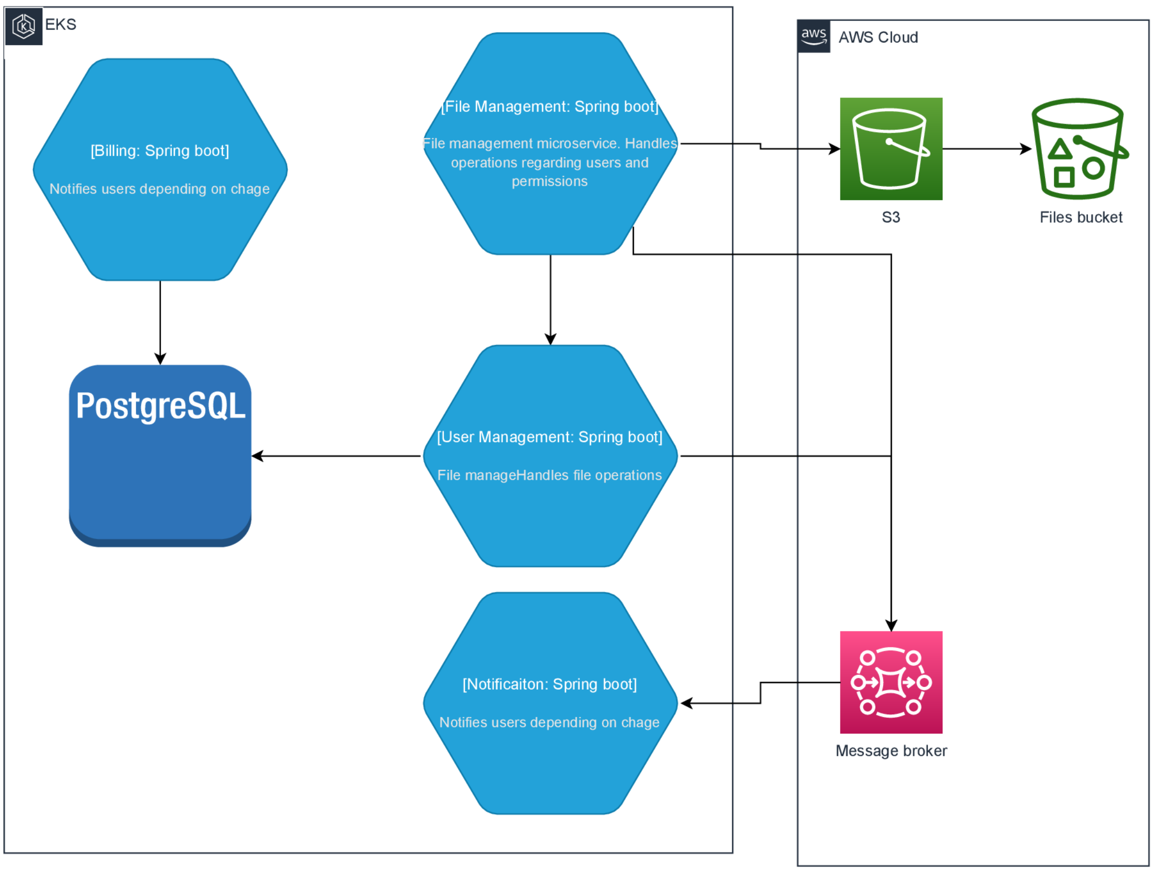

EKS
[Billing: Spring boot]
Notifies users depending on chage
[File Management: Spring boot]
File management microservice. Handles operations regarding users and permissions
PostgreSQL
[User Management: Spring boot]
File manageHandles file operations
[Notificaiton: Spring boot]
Notifies users depending on chage
AWS Cloud
S3
Files bucket
Message broker

Figure 5.2: AWS design

The following sections will go through the design process and explain the reasoning behind each component.

File storage

The first design was based on what was needed to be done regarding the actual files. The design was done under the assumption that the data consists of smaller files (up to 5Mb), as they can be kept in memory before sending them to the object storage. If the files were bigger, there would be a different approach used, where the component would become a stateful one, which would enable the files to be saved to a file storage that is mapped to the component. This would allow the download to resume in cases where there is interference that causes the transfer to crash. In case this would not be performed, the user would have to upload the same file from the beginning. Another reason is that if each file were of bigger size (at least 1GB), there would a problem to load them into the memory all at once. This would also add another level of complexity where the partially uploaded file would need to be handled in case where the user does not resume the upload. As the plan is to handle smaller file sizes, the option for partial updates is not as critical since the chance that the transfer fails is smaller and even if it fails, the time required for uploading is small. The storage type that will be used is object storage 4.2 as there is no need for the complexity of the standard file storage and no use for the advantages of the block storage that works better with big files and a lower volume of data. With all of this established, a file management component was created. This component would handle the upload of files and update the metadata for files.

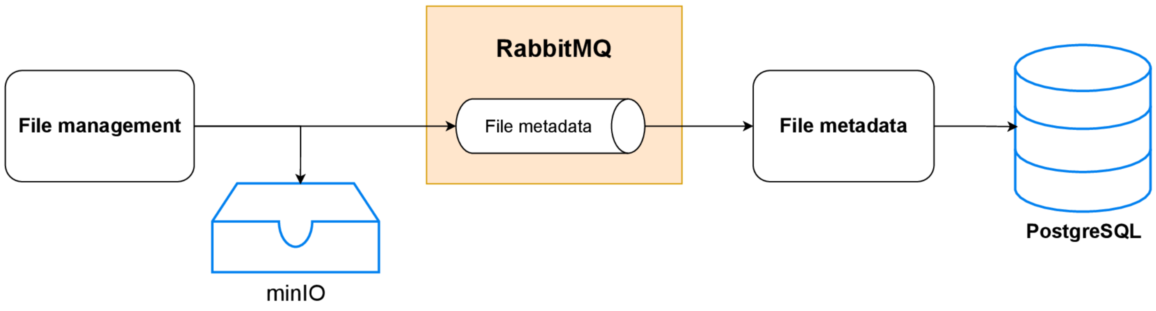


Figure 5.3: Architecture of the storage solution

User management

The second challenge of the application is how to handle users. When it comes to the management of users, we need to establish the complexity between changing the permissions and sharing files for users. There are 3 approaches possible that were specified in the section 4.5 for multitenant system.

1. In the first approach, we will be to have a database with separate schemas that will handle access per tenant. This will reduce the resources but make for a more complicated design as we need separate queries and access to schemas.
2. In the second approach, we will use separate databases. The expectation is to see an increase in resources utilization as more instances are active.
3. The last approach will be to have shared schemas but have a tenant identifier in the tables schema.

In the application, there is no need to differentiate between tenant. However, a solution that experiments with this will also be tried. For testing purpose, it will be slightly readjusted so that it is testable. As such, the best approach is to choose the last approach. If sign-in with external sources like google, twitter or facebook is included, then it would be better to use one of the other approaches as they would have different models that are used for authentication. These approaches do not transfer the user name and password, instead they use a securely transferred code that is based on your account.

Aside from the multitenant use case that we can show on the user management, it will also serve as the microservice where the scaling will be the most verified. The reason for that is that it interacts or provides data to many other services so if this microservice has some problems, there is a higher chance to see it propagate to other microservices.

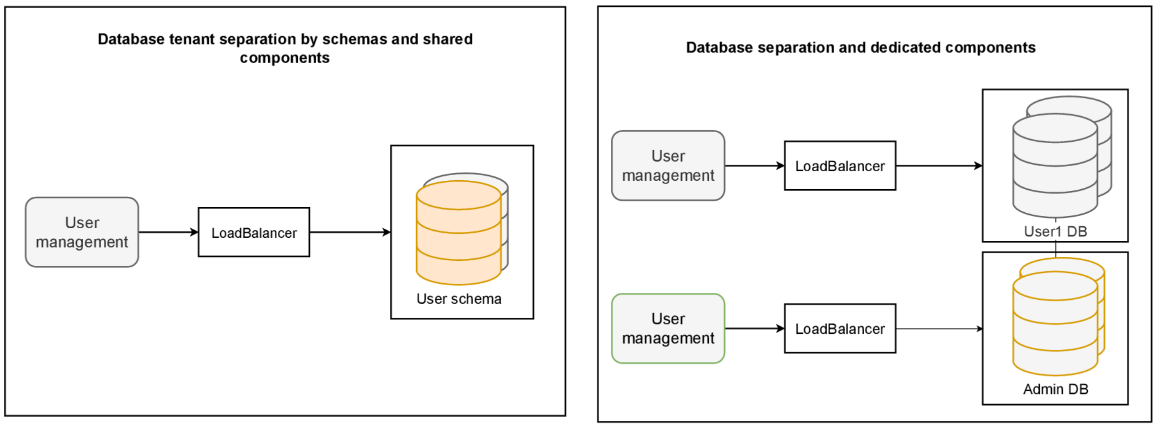


Figure 5.4: Multi tenant solutions. Left we have shared application logic and schema based DB, right is dedicated components and DB isolation, the approach for shared component and shared database with tenant identification field is not displayed

Database solution

Another part of the application is the database. The relational database will not be that different from how it is normally used. A load balancer is attached to the database which will send the requests to the correct databases. One of the important things is to divide the replicas of the database into read only and read/write databases. If this division is used, it improves the load on the databases. With this, one database replica becomes the source of truth while the other two will get the data from the main write database.

Communication between services

Another challenge that needs to be solved is how the microservices will communicate with each other. There are 2 main types that will be used: synchronous and asynchronous communication. Synchronous communication will be performed using restful APIs and asynchronous communication is done using the message broker.

Serverless

Serverless will be in an interesting position as its main purpose would be evaluating the changed load on the cloud after removing certain components. The components that will be analyzed are the storage where the AWS S3 object storage is used as a substitute for MinIO. Another component will be message broker where AmazonMQ substitutes the RabbitMQ but only partially as the RabbitMQ implementation that AmazonMQ provides is used.

Chapter 6

Implementation

In this chapter, we will go into the actual code implementation, show how the code looks like to create the needed services and how it is all configured. We will separate this first into components and establish what they should contain and then go into each microservice and how the used components are needed and utilized there. In the microservice part, the behaviour that is expected from each microservice is defined. Moreover, the interfaces that they provide are listed to further show what resources will be used.

6.1 Components implementation

The following section will describe the basic configuration needed to realize the patterns mentioned in chapter 3. We will go through the implementation in Kubernetes and it will be explained what each field is used for.

6.1.1 Load balancers/Services

The service is used as a separation layer, between the pods and the rest of the environment, which manages the traffic. This allows pods to be created dynamically with different names while all other microservices communicate with the pods through the service. The service layer can also be used as the load-balancer which handles the incoming traffic to the pods. With this, it is made sure that all applications can only see the service and not the pods. It also allows us to efficiently handle dynamic scale-up/scale-down as the service handles all the traffic and knows where the pods are. The following is a basic template for a service in Kubernetes.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: <service-name>
spec:
  selector:
    app: <service_selector>
  ports:
    - name: http
      port: <container_port>
      targetPort: <target_port>
  type: <service_type>
```

The spec section defines the specifications for the Service resource. The selector section specifies the label selector for the Service resource, which is set to match pods with labels app: <service_selector>. This selector is used to associate the Service resource with the pods that provide the service.

The ports section defines the port on which the service listens and which port it exposes. There are 2 values the name identification for the port, the port that is accessible in the container and the target port that will be available in the service and exposed to other services.

At the end, the type field specifies the type of the Service resource. In our solution, this is used to establish load-balancers for deployments but also classic services.

6.1.2 Deployments

In the demo application, deployments are used to establish the basic setup of the microservice. Deployments provide a simple way to update the application without downtime, by creating a new ReplicaSet with the updated Pods, and then gradually scaling down the old ReplicaSet and scaling up the new one. With Deployments, it is possible to roll back to a previous version of the application if a new release has issues. Deployments also enables performing rolling updates and rollbacks, which means that it is possible to update the application in a controlled and automated way, without risking any downtime or disruptions.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: file-management
spec:
  replicas: <number_of_replicas>
  selector:
    matchLabels:
      app: file-management
  template:
    metadata:
      labels:
        app: file-management
    spec:
      containers:
        - name: file-management
          image: matthew9164/filemanagement:1.0.5
          ports:
            - containerPort: 8090

```

As it is in service here, also the spec section defines the specification of the deployment. This will be the case in other Kubernetes components. The replicas field sets the number of initial replicas of the deployment. The selector section specifies the label selector for the deployment resource.

The template section provides a template for creating the pods that the Deployment will manage. The metadata field defines the labels for the pod. The spec field specifies the containers running within the pod. In our case, it will specify the image that will be used and which ports are available internally in the container.

6.1.3 PVC

Persistent Volume Claim[1] (PVC) is a request for a specific amount of storage to be provisioned dynamically by a storage provider. PVCs abstract the underlying storage medium, such as a physical disk or a cloud storage service, from the Pod that uses it, allowing for easy migration of Pods between nodes and clusters.

PVCs provide a way to manage storage resources in a decoupled manner from the Pod, so that the Pod can access storage without being aware of the underlying storage infrastructure. PVCs are used to define the requirements for the storage needed by a Pod, such as the access mode (read/write), storage class name, and size.

When a Pod requests a PVC, Kubernetes looks for an available PV that matches the requirements specified in the PVC. If a suitable volume is found, Kubernetes binds the PVC to the volume, and the volume is mounted into the Pod. If no suitable volume is found, Kubernetes dynamically provisions a new PV that matches the requirements specified in the PVC.

PVCs are essential for stateful applications, such as databases, that require persistent storage across Pod restarts and rescheduling. PVCs also provide a way to manage storage resources in a scalable and dynamic way, without requiring manual intervention.

In summary, PVCs provide a way to manage storage resources in a decoupled and dynamic manner in Kubernetes. They allow Pods to access storage without being aware of the underlying storage infrastructure, and they provide a scalable and dynamic way to manage storage resources in a Kubernetes cluster.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  # Name the volume chain
  name: db-persistent-volume-claim
spec:
  storageClassName: manual
  accessModes:
    # Allow ReadWrite to multiple pods
    - ReadWriteMany
  # PVC requesting resources
  resources:
    requests:
      # the PVC storage
      storage: 8Gi
```

```yaml
apiVersion: v1
# Kind for volume chain
kind: PersistentVolume
```

[1]https://kubernetes.io/docs/concepts/storage/persistent-volumes/

```yaml
metadata:
  # Name the persistent chain
  name: postgresdb-persistent-volume
  # Labels for identifying PV
  labels:
    type: local
    app: postgresdb
spec:
  storageClassName: manual
  capacity:
    # PV Storage capacity
    storage: 8Gi
  # A db can write and read from volumes to multiple pods
  accessModes:
    - ReadWriteMany
  # Specify the path to persistent volumes
  hostPath:
    path: "/data/db"
```

The PV file contains the field kind that represents the type which is PersistentVolume. The metadata section provides a name for the resource and labels for identification purposes, such as type: local and app: postgresdb.

The spec section defines the specifications for the PersistentVolume resource, including the storageClassName as manual and the capacity as 8GB. The accessModes are set to ReadWriteMany, which indicates that the volume can be read and written by multiple pods. Finally, the hostPath section specifies the path to persist the volumes, which is set to „/data/db“. Overall, this file defines a persistent volume that can be used by multiple pods for read and write operations and stores data in the specified path.

6.1.4 Rbac

RBAC allows administrators to define roles and permissions for users or groups of users, which are then used to restrict access to resources within the cluster. RBAC is used to limit access to resources such as Pods, Services, ConfigMaps, and Secrets, as well as to Kubernetes APIs and commands.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

The roleRef section defines the ClusterRole resource to bind to, which is a built-in cluster-admin role with an apiGroup of rbac.authorization.k8s.io and a kind of ClusterRole. The subjects section specifies the user to whom the role is being bound, which is a ServiceAccount named admin-user within the kubernetes-dashboard namespace. Overall, this file grants cluster-admin permissions to the specified ServiceAccount within the given namespace.

6.1.5 Secrets

Secrets are stored as base64-encoded data in Kubernetes, and are mounted as volumes or defined as an environment variables into the Pod that needs to access the secret. Secrets are used to transfer the needed credentials or data that should not be easily visible and it is recommended to save secret separately from the rest of the deployment. For our use case, where security of these credentials is not in scope of this thesis, it is kept in the same location for simplicity and visibility.

```yaml
apiVersion: v1
kind: Secret
metadata:
    name: minio-root-user
data:
  root-user: dXNlcg==
  root-password: cGFzc3dvcmQ=

```

Secret object is named „minio-root-user“. The Secret contains two pieces of sensitive information: a root user and a root password for a system or application.

The metadata section includes the name of the Secret, „minio-root-user“, and other optional information such as labels that can be used to organize and categorize the Secret.

The data section includes the base64-encoded string representation of the root user and root password values. In this example, the root user is „user“ and its base64-encoded value is „dXNlcg==“. Similarly, the root password is „password“ and its base64-encoded value is „cGFzc3dvcmQ=“.

6.1.6 Helm charts

Helm charts are a Kubernetes package manager that uses YAML files to define the various resources required to run an application, including deployment files, service files, and config maps. In addition, Helm charts contain templates that allow for customization of these resources based on specific values or parameters. Helm charts provide several advantages, including simplified management and upgrading of complex applications through the ability to package all necessary resources in a single chart. They also offer customization of Kubernetes resources through templates and a built-in dependency management system. However, Helm charts may be more complex to create and maintain compared to custom deployment files, especially for simpler applications. They also add an additional layer of complexity to the infrastructure, which can make issue resolution and problem debugging more challenging.

6.2 Microservices

In this section, we will go through the microservices and what we expect from them. First, we will shortly look at the implementation and introduce the data they work with and how they do it. The custom microservice that are created are written in Java using the spring boot framework[2]. They have a standard separation into layers which may seem similar to the N-tier architecture3.2. One of the notable technologies for this is the `spring cloud` library, which could enable a more dynamic configuration of the microservices and has already implemented some of the features mentioned in the previous sections(circuit breaker, service-to-service calls, leadership election...), but as this concepts are hidden in the library and automatically configured when using some of the provided function, it was decided not to obfuscate the functionality. So more simple frameworks were used for communication and the patterns were either implemented explicitly (circuit braker) or are present in Kubernetes deployment are therefore are not language bound.

6.2.1 SQL database

The relational database used in the demo application is PostgreSQL[3]. An out-of-the box solution is used since we need only minimal configuration. As for the actual design of the table, a very simple design with 2 tables is used. The design can be seen in the following figure 6.1.

This database will be used for saving the users in our application. The other purpose will be to save some metadata about files. Although there is an option to save these metadata with files to our object storage as both MinIO and S3 support custom metadata, but they require to download the whole file to access them. With this in mind, the metadata is saved to the database and all operations pertaining files will have to provide correct propagation in case when the files from the storage are deleted or updated in any way.

6.2.2 User management

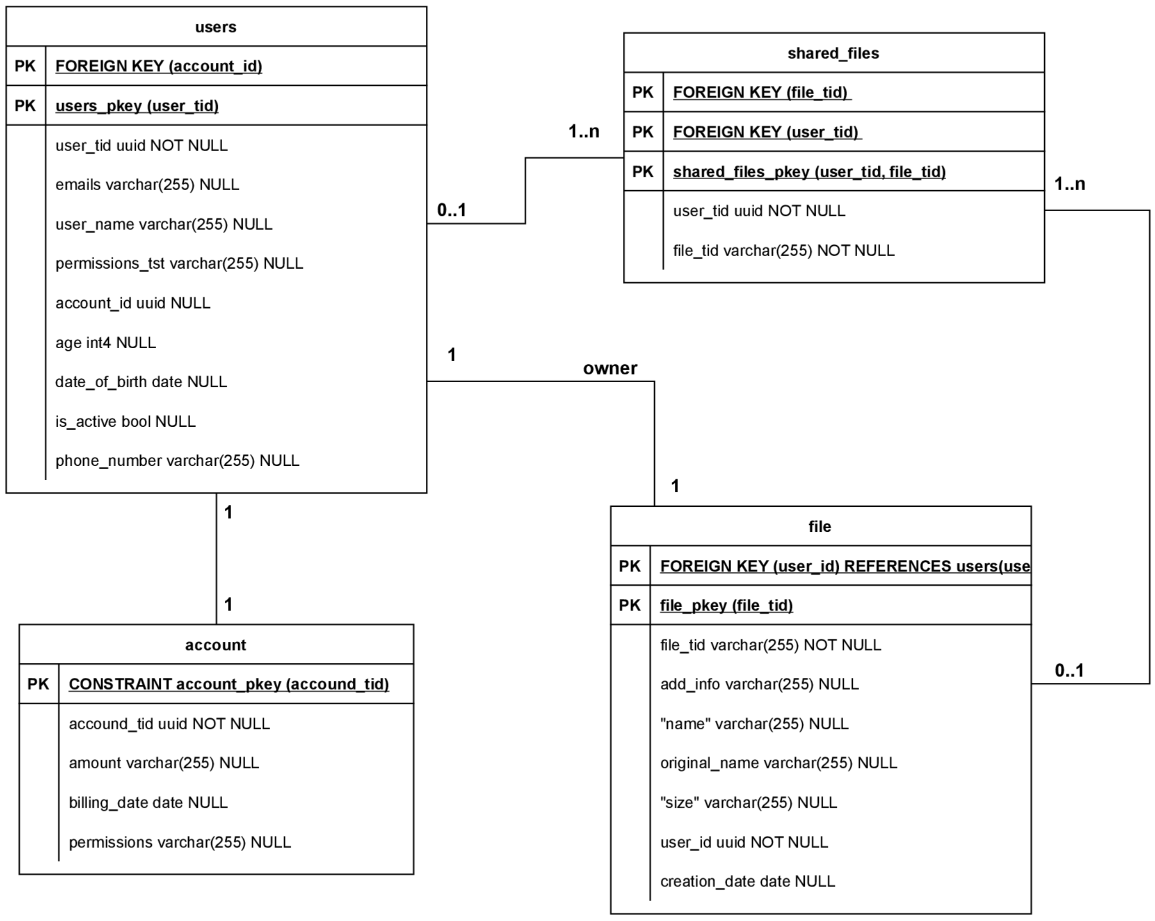
The user management is our custom microservice which will handle requests pertaining to user operations. It will also be used in the use case where files are shared between users and give the correct permissions to files to the correct user. This microservice will be used to showcase proper scaling up or down depending various criteria such as CPU usage and memory usage. It will also show the multitenant approach 4.5 and while it was needed it in our example application it will showcase the benefit and drawback of the pattern.

The implementation is done in Java using the spring boot framework. The microservices provides an API interface for communication and needs active connections to the database and RabbitMQ.

The interface is accessible using swagger offers the following operations

- `POST /users` – create a new user
- `GET /users/id` – get user with concrete id
- `GET /users` – get list of all users
- `PUT /users/id` – update user with id

[2]https://spring.io/
[3]https://www.postgresql.org/docs/15/index.html


users
PK FOREIGN KEY (account_id)
PK users_pkey (user_tid)
user_tid uuid NOT NULL
emails varchar(255) NULL
user_name varchar(255) NULL
permissions_tst varchar(255) NULL
account_id uuid NULL
age int4 NULL
date_of_birth date NULL
is_active bool NULL
phone_number varchar(255) NULL
shared_files
PK FOREIGN KEY (file_tid)
PK FOREIGN KEY (user_tid)
PK shared_files_pkey (user_tid, file_tid)
user_tid uuid NOT NULL
file_tid varchar(255) NOT NULL
1..n
0..1
1..n
1
owner
1
1
1
account
PK CONSTRAINT account_pkey (accound_tid)
accound_tid uuid NOT NULL
amount varchar(255) NULL
billing_date date NULL
permissions varchar(255) NULL
file
PK FOREIGN KEY (user_id) REFERENCES users(use
PK file_pkey (file_tid)
file_tid varchar(255) NOT NULL
add_info varchar(255) NULL
"name" varchar(255) NULL
original_name varchar(255) NULL
"size" varchar(255) NULL
user_id uuid NOT NULL
creation_date date NULL
0..1
Figure 6.1: ER diagram for database. Database is created in user management

- `DELETE /users/id` – delete an user with having given id
- `PUT /users/filesId/share/userId` – share a existing file with an user
- `POST /users/share/userId` – create an entry for file that is saved in the storage. The userId is used to identify the owner

6.2.3 File management

File management will also showcase proper scalability on increasing traffic load. It will be also the microservice that is responsible for handling the upload of the files to the storage. This will also showcase the circuit breaker pattern where in case where the s3 download does not work it will upload to a backup storage provider where it will be saved for a limited amount of time.

The implementation is done in Java using the spring boot framework. The microservices provide an API interface for communication and need active connections to the storage and RabbitMQ.

The interface is accessible using swagger offers the following operations:

- `POST /files` – save file to storage and propagate changes to database

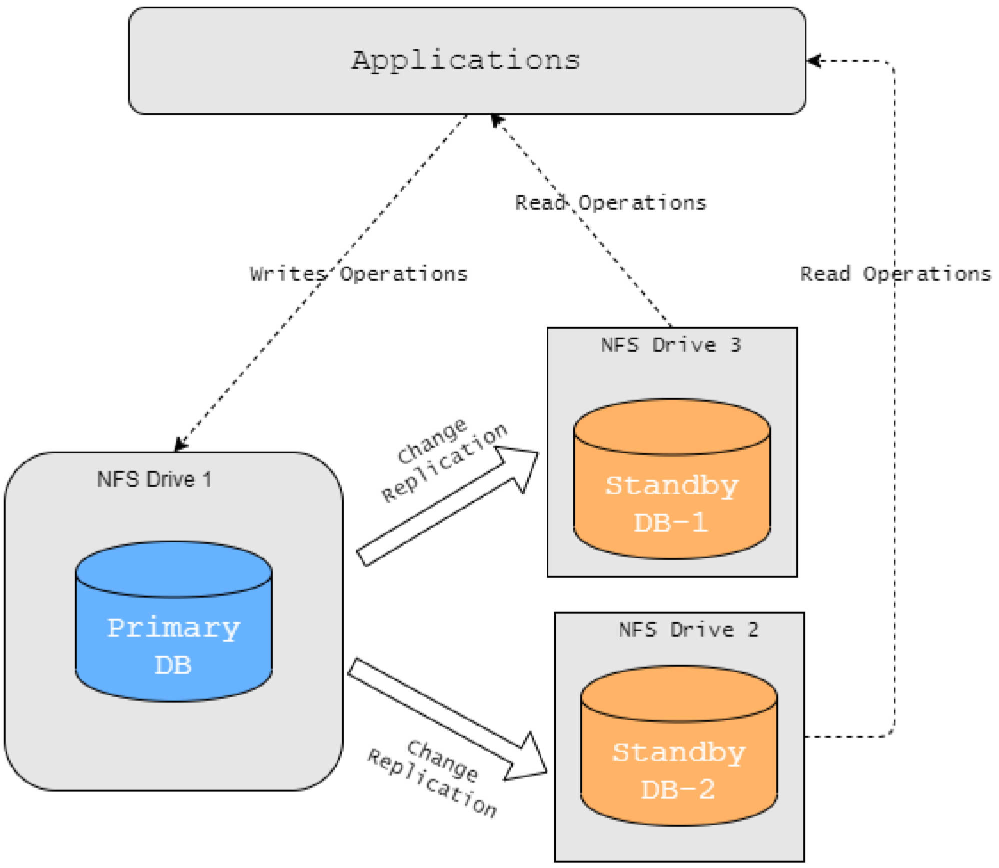


Figure 6.2: Database setup primary with read only replicas

- `GET /files/id` – retrieve file from storage with the given id
- `DELETE /files/userId` – delete file from storage and propagate change to user management to update the reference in database

6.2.4 Billing

The billing microservice will occur on timer based events. The design of this microsevice should behave with the predictable load in mind. As such, there will be an automatic job that will scale the microservice down and up every day. The microservice will run outside of the usual peak hours, which is between 1 to 4 am. During this time, it will operate normally where it gets the user information and calculates bills for users. This calculation functionality is just mocked as it is outside the scope of this thesis to send actual emails with bills to users email addresses. The microservice starts with a call to the endpoint from the job to start processing the billing.

6.2.5 Notification

The notification microservice will serve as the main microservice to show the benefit of asynchronous communication using message brokers which in our case is done using RabbitMQ. The normal behaviour of the microservice is to be connected with a listener to an existing queue where it waits for a message. The default behaviour of RabbitMQ is message delivery with a round robin system. Thanks to this, even if the application is scaled up and there are multiple listeners to the queue, only 1 single message is delivered and there are no duplicate messages. After receiving the message, the microservice should send an email notification to the user that a file was shared with him. This functionality is also mocked, same as in billing, as there is no need to send the actual email to show this behaviour.

6.2.6 RabbitMQ

RabbitMQ[4] is used as our message broker. It makes it possible for us to decouple the communication efficiently while also serving as a temporary storage for messages. The deployment will be done with existing with AmazonMQ on AWS and with custom deployments of RabbitMQ. As in the case with the storage, the AWS solution runs outside of the environment and therefore decreases the load that is run on the cluster. The created resources in RabbitMQ can be seen in the following table 6.1. The queues, as can be seen from their name, are going to be used for notifications or file data.

Exchange name	**Exchange type**	**Queue name**	**Queue type**
file-metadata-exchange	topic	file.queue	classic
notification-exchange	topic	notification.queue	classic

Table 6.1: Created exchanges and queues in RabbitMQ

6.2.7 Storage

The storage solution will be between two types of object storage. The first one is MinIO[5], which will be deployed same as the other microservices. The other solution is S3[6], which will be used on AWS to see how it compares to MinIO and to see the benefit of cloud hosted services provided by the solution provider.

[4]https://www.rabbitmq.com/documentation.html
[5]https://min.io/docs/minio/kubernetes/upstream/
[6]https://aws.amazon.com/s3/

Chapter 7

Experiments

This chapter will focus on the experiments that should verify if the applied patterns had the expected result. We divided the interesting patterns by the following characteristics: Performance, Resource consumption, Scalibility, Fault tolerance and one more general category where we experiment with special patterns. The experiments will be performed on one node and all services will run on that node. The only exception is when we run AmazonMQ and S3 as they will represent the serverless architecture in this example. For the metrics, we will mainly look at CPU and memory load. There are other metrics like storage space utilization or latency but there was no interesting performance difference to include them.

First, it is needed to be defined how these metrics are obtained. For this purpose, the Kubernetes `Metrics API` is used. This API is also needed for the HPA as they depend on the measured metrics. The commands to see the current resource consumption are the following:

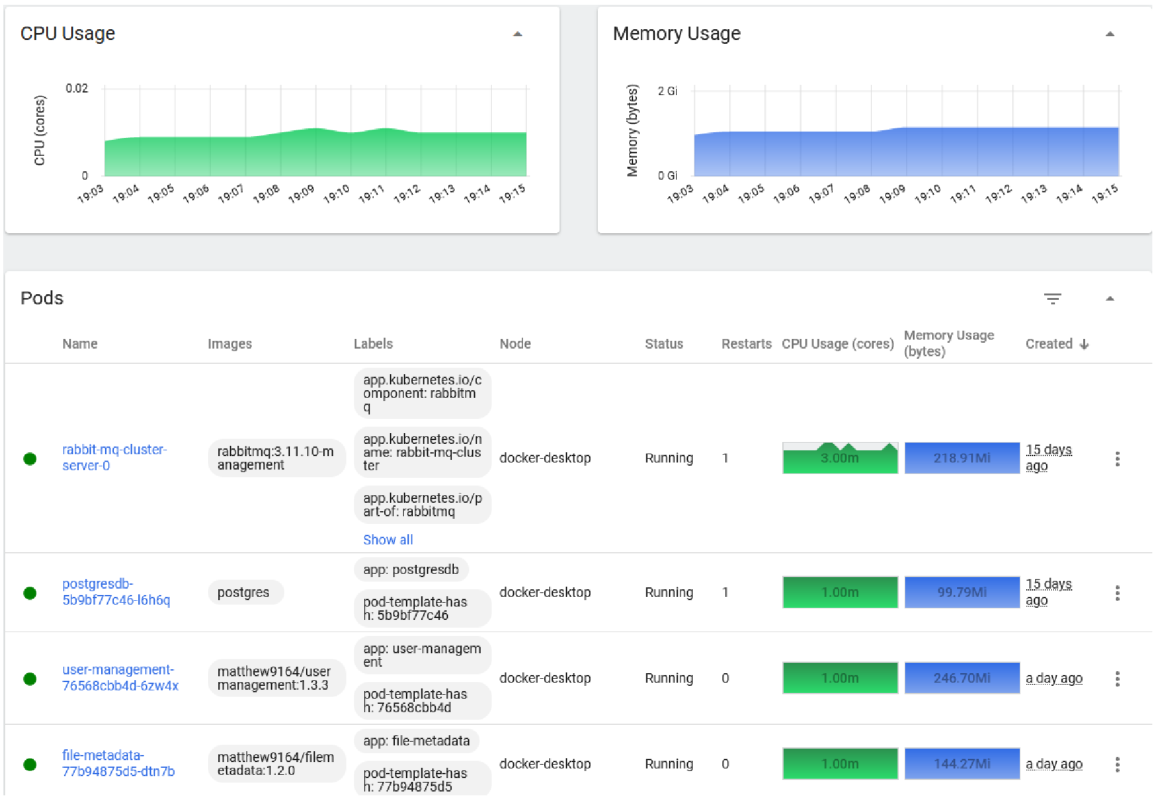
```
$ kubectl top node
NAME             CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
docker-desktop   275m         1%     6311Mi          39%

$ kubectl top pod file-management-7bb9cbf564-f55tt
NAME                                CPU(cores)   MEMORY(bytes)
file-management-7bb9cbf564-f55tt    1m           153Mi
```

Another view that is used is the kubernetes-dashboard[1], which provides a better view for the cluster along with a timeline for the resource consumption on the node and pods.

To actually be able to tell if the experiments perform how it is expected, there is a need to know the CPU and memory usage in the default state. Limits to these resources have to be set so that no noisy-neighbour effect occurs where one service will start to fail because other services used all available resources before it could access them. As can be seen in table 7.1, in the default state with no load, there is only negligible CPU usage. The only one with an increased load is the message broker which requires more CPU to operate. However, even in this case, the usage is still negligible. On the memory side, the custom microservices required somewhere between 300-400 MB, which is expected for a Java spring boot application. On the other hand, RabbitMQ only had a memory usage around 150 MB while the expected usage is in the range of 256-512 MB. After an investigation, it was found out that this is mostly due to the fact that not many plugins are loaded and only a few queues are needed in our application.

[1]https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/



Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↓
●	rabbit-mq-cluster-server-0	rabbitmq:3.11.10-management	app.kubernetes.io/component: rabbitmq app.kubernetes.io/name: rabbit-mq-cluster app.kubernetes.io/part-of: rabbitmq Show all	docker-desktop	Running	1	3.00m	218.91Mi	15 days ago
●	postgresdb-5b9bf77c46-l6h6q	postgres	app: postgresdb pod-template-hash: 5b9bf77c46	docker-desktop	Running	1	1.00m	99.79Mi	15 days ago
●	user-management-76568cbb4d-6zw4x	matthew9164/usermanagement:1.3.3	app: user-management pod-template-hash: 76568cbb4d	docker-desktop	Running	0	1.00m	246.70Mi	a day ago
●	file-metadata-77b94875d5-dtn7b	matthew9164/filemetadata:1.2.0	app: file-metadata pod-template-hash: 77b94875d5	docker-desktop	Running	0	1.00m	144.27Mi	a day ago

Figure 7.1: Kubernetes dashboard view with available metrics. Deployed on local machine.

	CPU%	MEMORY
File management	<1	~153 MB
File metadata	<1	~144 MB
User management	<1	~200 MB
Notifications	<1	~82 MB
Billing	<1	~110 MB
PostgreSQL	<1	~100 MB
RabbitMQ	<2	~150 MB
Storage	<2	~100 MB

Table 7.1: Default usage of CPU and memory

As such the following memory constraints are set on the resources:

- File management – min 256 MB, max 512 MB
- User management – min 256 MB, max 512 MB
- Notifications – min 256 MB, max 512 MB
- Billing – min 256 MB, max 512 MB
- PostgreSQL – min 64 MB, max 256 MB
- RabbitMQ –min 256 MB, max 512 MB

- Storage –min 256 MB, max 512 MB

The reason for the low and maximum constraint is to reduce the effect between microservices and for our purpose, there is not much difference between 1GB a 512MB usage as the interesting behaviour occurs when the microservices reach their limit. To make sure that the Java Memory Model does not interfere with the memory consumption during the testing of the memory lead, the garbage collector is set to the Epsilon GC[2]. The used flags to make this possible were `-XX:+UnlockExperimentalVMOptions` and `-XX:+UseEpsilonGC`.

Scaling

The first experiment will be based around the proper scaling in the cloud environment. For this, the user management will be experimented with where there are 2 endpoints which are used to either test memory usage or increase the CPU load. For the testing itself, there are going to be GET requests which are sent to the endpoints of the application. These GET requests are send to 2 endpoints that are primarily for increasing memory and one for increasing CPU usage. The rate of the requests will be increased overtime to reach the limit. The maximum scaling is set to 5 pods with the default being 3 pods. The monitored resources which should be reached are CPU usage of 82% and memory usage of 82%. The first test is without the autoscaler enabled to ensure that there is enough requests produced so that the applications limits are reached and that scaling is needed. The result without the scaling can be seen in the following table 7.2 with the peak values that were reached. The result were as expected, the memory load caused the application to crash, which is the behaviour of the epsilon garbage collector when it runs out of memory. The CPU test caused quite a significant increase in memory along with the CPU intensive operation and this is again due to the garbage collector that was set which reduced the option for memory optimization.

	CPU%	MEMORY%
Memory test	31%	100%
CPU test	100%	93%

Table 7.2: Performance for tests with no scaling

Now with the HPA 4.1 autoscaler up and running, the deployment was able to handle the previous load. While the pods could still crash, it requires a much higher rate of requests(around a 200% increase) to be sent to the pod. While it now was able to handle the rate of requests, one problem that was observed is that when the load increased too quickly, there was still a chance for the pods to crash as the needed pod did not start the new instance of the microservice. Due to this, it was essential to slowly increase the load and not overload it at once. The number of requests should not exceed 100 sent requests per second but this is specific to the environment. This problem could have been avoided with better garbage collecting as our garbage collector really impacted this problem. With the same load that caused the crash under the epsilon gc, the standard serial garbage collector or the G1 garbage collector could handle the memory way more efficiently and such a crash did not occur. The results can be seen in table 7.3 with the values being after the scaling took place. One small detail in the table is even though the scaling threshold was set to

[2]https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector

80% of memory usage, the result exceeded it to 82%. This happened due to the upper limit for replicas that was reached.

	CPU%	MEMORY%
Memory test	3%	82%
CPU test	72%	76%

Table 7.3: Performance for tests with scaling. Number of pods increased from 3 to 5

To be absolutely sure that the scaling did happen because the target utilization was hit, the log of the autoscaler can be used where the message describing the change is present. This can be seen in the following log.

```
#kubectl describe user-management-hpa

Conditions:
    Type            Status  Reason              Message
    ----            ------  ------              -------
    ScalingLimited  False   DesiredWithinRange  the desired count is within
                                                the acceptable range
Events:
    Reason               From                       Message
    SuccessfulRescale   horizontal-pod-autoscaler  New size: 5;
        reason: cpu resource utilization (percentage of request)
                above target
```

Failure handling

Another part to test is the ability of the application to handle failure of components. This will showcase the usage of the message broker who decouples the dependencies between services and enables asynchronous communication. This was very apparent when one service was overloaded while the other was still managing the traffic. If the overloaded microservice was a listener to a queue in RabbitMQ, it would still be able to process all the requests as they are saved in the queue. The requests were saved to RabbitMQ and as such the microservice that was not able to keep up had enough time to process all requests. The file management metadata microservice will be used to demonstrate this pattern of decoupling. The exact use case is when a new file is created, it is saved to the storage and a message is sent through RabbitMQ to update the metadata in the database. There is the file metadata service, which is created for the purpose of handling this data. If it was sent directly to the metadata service and the service would be unavailable, then a backup plan is needed to decide where to save this data or a different way of how to handle the request. With this approach, it is sent to the message broker who keeps the message until the microservice is up, thus not impending the application as a whole. But this also opens another question. What to do when the message broker is also down? The advantage is that, on average, the message brokers are built to be high-availability and with this characteristic, they should provide more stability. But if a crash still occurs, there are many ways to handle this. One is to have some local cache in the microservice where the data is saved. Another one is to have a circuit breaker which either fails-fast or has a backup method. In our case, it could be having a direct connection to the metadata service in case of this failure. But the developers would have to be careful to not misuse this connection and only use this in case of emergency. For simplicity, the fail-fast approach was chosen as the back up connection is straightforward and would not needlessly complicate the setup in our case.

The experiment will first test what would happen if the file metadata service would not be available and then what would happen when the message broker is not available.

Multitenancy pattern

The multitenancy pattern in cloud is quite interesting. The user management microservice was deployed as shared where there are three shared components and the isolation was made on the database layer and another version where the components are isolated and deployed as three separate services while there is also separation in the database on a specific tenant id. Under normal load where there is no need for scaling, the performance is nearly identical with the dedicated components performing slightly better. This was mostly due to some small differences in the implementation and the need to differentiate between requests in the shared component. This can be seen in the following table 7.4.

	CPU%	MEMORY%
Dedicated component	18%	72%
Shared component	20%	68%

Table 7.4: Multi tenancy testing with under normal load.

The interesting situation happens when the load is increased, but just for one tenant. The shared components behaved materially better as the load was properly distributed. On the other hand, for the dedicated components, while two of the pods were with similar usage as before, one pod needed to scale up, increasing the number of pods from three to four. Due to the overhead need in the new pod, a bigger resource consumption was observed. This can be seen in the table 7.5.

	CPU%	MEMORY%
Dedicated component	23%	72%
Shared component	21%	56%

Table 7.5: Multi tenancy testing under load. Number of pods increased from 3 to 5.

Serverless

To properly show the serverless function on the cloud, the storage and message broker are used. The MinIO storage will be swapped for the AWS solution S3 and RabbitMQ will be swapped for AmazonMQ, which still gives the option between ActiveMQ and RabbitMQ. The most important part for this is the comparison of how much the load on the node is impacted and also how the latency increases when the solution is moved to another location. While in private cloud, everything from PVC, PV to the storage providers had to be set up. In AWS, there is just some need for manual configuration of the S3 object storage and AmazonMQ. There is also the responsibility to establish an identity provider so that the microservices in that are in AWS EKS[3] can communicate with S3 and AmazonMQ. One

[3]https://aws.amazon.com/eks/

metric that was found to be of interest was the latency change when the services are moved to the AWS provided solutions. But after a few experiments, there was no material increase in latency, so with that in mind, latency was removed from the table. This was done by verifying the latency to the cloud by sending HTTP requests from the local machine and also from inside the cloud. The main point to consider is that it was still in the same availability zone.

The result with the system in the default state did not change much in terms of memory percentage used but did change in the total consumption of memory as when we combine the set maximum for both services, the saved resources that were obtained without the system even being active are around 250 MB. This can be seen in the table 7.6.

	CPU%	MEMORY%
In cloud	1	72
AWS solution	1	65

Table 7.6: No load in cloud

As when there was actual load on the system, then the result became more obvious, which can be seen in table 7.7.

	CPU%	MEMORY%
In cloud	34	89
AWS solution	13	72

Table 7.7: Simulating average load. Stats are for Kubernetes node where the pods are running.

Asynchronous pattern

To properly show one of the most important characteristics, the notification microservice will be used. For the experimenting, many files will be shared and a notification to inform the user is created. Thanks to the asynchronous decoupling that RabbitMQ provides, even in case of failure of the notification, microservice the flow is not interrupted. The same behaviour was observed if the notification microservice failed and was deployed again at a later time. The peak resources consumption can be seen in the following table 7.8.

	CPU%	MEMORY%
Notifications	23	78
User management	12	73
RabbitMQ	10	10

Table 7.8: Load while generating notification

Time based pattern

The last experiment will be with the billing microservice. The comparison will be between when the microservice is running without load and with load. Considering that it is active only for a few hours, the cost and resources that will be saved can be calculated precisely. It will be performed while showing the IDEAL properties 3.1 and the predictable workload pattern 4.1. It also showcases different patterns that previous microservices used as load balancers. If we count the default resource consumption of the service then the cost to run it would be from a memory point of view 100MB of taken up space. It would also use a very small percentage of CPU so that it can run. For the purpose of this experiment the time was changed to a more appropriate time and the microservice was left running only for 10 minutes before it was scaled down. The scale-up and scale-down worked correctly and it this case the optimization of resources is evident as the microservice did not take up any resources during the down time and the only needed resources are for the automated cron job for this service.

Chapter 8

Conclusion

The main objective of this thesis was to introduce the cloud environment challenges, explore a variety of solutions available from cloud providers, and analyse what are the best practices to effectively utilize these resources. Chapters 2, 3 and 4 introduce several approaches for handling the complex challenges that can occur in the cloud environment. One of the key insights gathered from this thesis is the importance of adopting a holistic approach to cloud environment design. A wide range of factors needs to be considered, such as workload requirements, resource requirements, and cost considerations, which can be adjusted to the specific needs for each user. With all these factors in mind, a proper design of the application is needed. Another key insight gained from these chapters is the value of leveraging automation and orchestration tools to simplify and automate the cloud management. By automating routine tasks and deploying resources with predetermined configurations that are standardized, the risk of fatal errors is reduced. With all this knowledge in mind, a demo application that should demonstrate some of the patterns that were investigated in this thesis was created. The tests mostly focused on scalability, failure handling and asynchronous communication. Different experiments were performed on this application to verify whether the patterns that are used in the application work correctly. These experiments helped to verify the patterns worked under our expectations and the demo application was published as open source.

As for future improvements that could be made, not all patterns were tested and there are certainly still some interesting use cases that can be experimented with. Due to time constraints, only the most essential metrics, such as CPU and memory, were monitored. While they are considered to be among the most important metrics, there is still a number of other interesting metrics that could be monitored and the chosen solutions all support prometheus integration for custom metrics. Aside from the cloud patterns, a comparison between providers and their solutions could be intriguing, as the performance increase when using AWS provided solutions was not negligible but users are billed for these service so they need to design their application with these costs in mind.

Bibliography

[1] *AWS Documentation* [online]. 2021 [cit. 2022-12-10]. Available at: https://aws.amazon.com/getting-started/cloud-essentials/?pg=gs.

[2] *Google Cloud overview* [online]. 2021 [cit. 2022-12-10]. Available at: https://cloud.google.com/docs/overview.

[3] *RabbitMQ* [online]. 2021 [cit. 2023-11-5]. Available at: https://www.rabbitmq.com/documentation.html.

[4] Aljahdali, H., Albatli, A., Garraghan, P., Townend, P., Lau, L. et al. Multi-Tenancy in Cloud Computing. In: IEEE. *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. April 2014, p. 344–351. DOI: 10.1109/SOSE.2014.50. ISBN 978-1-4799-3616-8.

[5] Alseelawi, N. S., Adnan, E. K., Hazim, H. T., Alrikabi, H. and Nasser, K. Design and implementation of an e-learning platform using N-TIER architecture. International Association of Online Engineering. 2020. ISSN 1865-7923.

[6] Anwar, A., Cheng, Y., Gupta, A. and Butt, A. R. Taming the Cloud Object Storage with MOS. In: *Proceedings of the 10th Parallel Data Storage Workshop*. New York, NY, USA: Association for Computing Machinery, 2015, p. 7–12. PDSW '15. DOI: 10.1145/2834976.2834980. ISBN 9781450340083. Available at: https://doi.org/10.1145/2834976.2834980.

[7] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S. et al. *Serverless Computing: Current Trends and Open Problems*. 2017.

[8] Brewer, E. CAP twelve years later: How the„ rules" have changed. *Computer*. IEEE. 2012, vol. 45, no. 2, p. 23–29. ISSN 1558-0814.

[9] Burns, B., Beda, J., Hightower, K. and Evenson, L. *Kubernetes: up and running*. 1st ed. „ O'Reilly Media, Inc.", 2022. 1–15 p. ISBN 9781491935675.

[10] Casalicchio, E. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*. Springer. 2019, p. 221–235.

[11] Chandra, D. G. BASE analysis of NoSQL database. *Future Generation Computer Systems*. Elsevier. 2015, vol. 52, p. 13–21. ISSN 0167-739X.

[12] Comer, D. E. *The Cloud Computing Book: The Future of Computing Explained*. Chapman and Hall/CRC, 2021. ISBN 978-0367706807.

[13] Curino, C., Jones, E., Popa, R., Malviya, N., Wu, E. et al. Relational Cloud: A Database-as-a-Service for the Cloud. In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. April 2011, p. 235–240. ISBN 978-9-5323-3050-2.

[14] Curry, E. Message-Oriented Middleware. In: Mahmoud, Q. H., ed. *Middleware for Communications*. Chichester, England: John Wiley and Sons, 2004, chap. 1, p. 1–28. DOI: 10.1002/0470862084.ch1. ISBN 978-0-470-86206-3. Available at: http://www.edwardcurry.org/publications/curry_MfC_MOM_04.pdf.

[15] Cusumano, M. Cloud computing and SaaS as new computing platforms. *Communications of the ACM*. ACM New York, NY, USA. 2010, vol. 53, no. 4, p. 27–29. ISSN 0001-0782.

[16] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F. et al. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*. Springer. 2017, p. 195–216.

[17] Etzkorn, L. H. *Introduction to Middleware: Web Services, Object Components, and Cloud Computing*. Chapman and Hall/CRC, 2017. 617–626 p. ISBN 978-1498754071.

[18] Fehling, C., Leymann, F. and Retter, R. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. ISBN 978-3-7091-1568-8.

[19] Gessert, F., Friedrich, S., Wingerath, W., Schaarschmidt, M. and Ritter, N. Towards a Scalable and Unified REST API for Cloud Data Stores. In: Plödereder, E., Grunske, L., Schneider, E. and Ull, D., ed. *44. Jahrestagung der Gesellschaft für Informatik, Big Data - Komplexität meistern, INFORMATIK 2014, Stuttgart, Germany, September 22-26, 2014*. GI, 2014, P-232, p. 723–734. LNI. ISBN 978-3-88579-626-8. Available at: https://dl.gi.de/20.500.12116/2975.

[20] Ghomi, E. J., Rahmani, A. M. and Qader, N. N. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*. Elsevier. 2017, vol. 88, p. 50–71. ISSN 1741-8577.

[21] Grolinger, K., Higashino, W. A., Tiwari, A. and Capretz, M. A. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: advances, systems and applications*. Springer. 2013, vol. 2, no. 1, p. 1–24. ISSN 2192-113X.

[22] Gupta, B., Mittal, P. and Mufti, T. A Review on Amazon Web Service (AWS), Microsoft Azure & Google Cloud Platform (GCP) Services. In: *Proceedings of the 2nd International Conference on ICT for Digital, Smart, and Sustainable Development, ICIDSSD 2020, 27-28 February 2020, Jamia Hamdard, New Delhi, India*. EAI, March 2021. DOI: 10.4108/eai.27-2-2020.2303255. ISBN 978-1-63190-292-5.

[23] Harrington, J. L. *Relational database design and implementation*. Morgan Kaufmann, 2016. ISBN 978-0128043998.

[24] Herbst, N. R., Kounev, S. and Reussner, R. Elasticity in cloud computing: What it is, and what it is not. In: Kephart, J. O., Pu, C. and Zhu, X., ed. *10th*

international conference on autonomic computing (ICAC 13). 2013, p. 23–27. ISBN 978-1-931971-02-7.

[25] Jatana, N., Puri, S., Ahuja, M., Kathuria, I. and Gosain, D. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*. Citeseer. 2012, vol. 1, no. 6, p. 1–5. ISSN 2278-0181.

[26] Jula, A., Sundararajan, E. and Othman, Z. Cloud computing service composition: A systematic literature review. *Expert systems with applications*. Elsevier. 2014, vol. 41, no. 8, p. 3809–3824. ISSN 0957-4174.

[27] Khiyaita, A., Bakkali, H. E., Zbakh, M. and Kettani, D. E. Load balancing cloud computing: State of art. *2012 National Days of Network Security and Systems*. IEEE. 2012, p. 106–109. DOI: 10.1109/JNS2.2012.6249253. ISSN 978-1-4673-1052-9.

[28] Lablans, M., Borg, A. and Ückert, F. A RESTful interface to pseudonymization services in modern web applications. *BMC medical informatics and decision making*. BioMed Central. 2015, vol. 15, no. 1, p. 1–10. ISSN 1472-6947.

[29] Leymann, C. F. F., Retter, R., Schupeck, W. and Arbitter, P. *Cloud computing patterns*. 2014. ISBN 978-3-7091-1568-8.

[30] Li, J., Wang, Q., Lee, P. P. and Shi, C. An in-depth analysis of cloud block storage workloads in large-scale production. In: IEEE. *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020, p. 37–47. ISBN 978-1-7281-7645-1.

[31] Martinekuan. *Microsoft Azure well-architected framework - azure architecture center*. 2022 [cit. 2022-12-10]. Available at: https://learn.microsoft.com/en-us/azure/architecture/framework/.

[32] Masdari, M. and Khoshnevis, A. A survey and classification of the workload forecasting methods in cloud computing. *Cluster Computing*. Springer. 2020, vol. 23, no. 4, p. 2399–2424. ISSN 1386-7857.

[33] Michelson, B. M. *Event-driven architecture overview* [online]. 2006 [cit. 11-5-2023]. Available at: https://elementallinks.com/el-reports/EventDrivenArchitectureOverview_ElementalLinks_Feb2011.pdf.

[34] Moldovan, D., Copil, G., Truong, H.-L. and Dustdar, S. MELA: Monitoring and Analyzing Elasticity of Cloud Services. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. 2013, vol. 1, p. 80–87. DOI: 10.1109/CloudCom.2013.18. ISBN 978-0-7695-5095-4.

[35] Moreno Vozmediano, R., Montero, R. S. and Llorente, I. M. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. *Computer*. 1st ed. IEEE. 2012, vol. 45, no. 12, p. 65–72. ISSN 0018-9162.

[36] Moyer, C. M. *Building Applications in the Cloud: Concepts, Patterns, and Projects*. Pearson Education India, 2011. ISBN 978-0321720207.

[37] Olowu, M., Yinka Banjo, C., Misra, S. and Florez, H. A secured private-cloud computing system. In: Springer. *International Conference on Applied Informatics.* 2019, p. 373–384. ISBN 978-3-030-32475-9.

[38] Pahl, C. Containerization and the paas cloud. *IEEE Cloud Computing.* IEEE. 2015, vol. 2, no. 3, p. 24–31. ISSN 2325-6095.

[39] Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing.* IEEE. 2017, vol. 7, no. 3, p. 677–692. ISSN 2168-7161.

[40] Pahl, C. and Lee, B. Containers and Clusters for Edge Cloud Architectures – A Technology Review. In: *2015 3rd International Conference on Future Internet of Things and Cloud.* August 2015, p. 379–386. DOI: 10.1109/FiCloud.2015.35. ISBN 978-1-4673-8103-1.

[41] Rad, B. B., Bhatti, H. J. and Ahmadi, M. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS).* International Journal of Computer Science and Network Security. 2017, vol. 17, no. 3, p. 228–232. ISSN 1738-7906.

[42] Shan, T. C. and Hua, W. W. Solution architecture for n-tier applications. In: IEEE. *2006 IEEE International Conference on Services Computing (SCC'06).* 2006, p. 349–356. ISBN 0-7695-2670-5.

[43] Singh, A. and Chatterjee, K. Cloud security issues and challenges: A survey. *Journal of Network and Computer Applications.* 2017, vol. 79, p. 88–115. DOI: https://doi.org/10.1016/j.jnca.2016.11.027. ISSN 1084-8045. Available at: https://www.sciencedirect.com/science/article/pii/S1084804516302983.

[44] Siriwardana, P., Fremantle, P., Azeez, A., Leelaratne, D., Perera, S. et al. Multi-tenant SOA Middleware for Cloud Computing. In: *2013 IEEE Sixth International Conference on Cloud Computing.* Los Alamitos, CA, USA: IEEE Computer Society, Jul 2010, p. 458–465. DOI: 10.1109/CLOUD.2010.50. ISBN 978-0-7695-5028-2. Available at: https://doi.ieeecomputersociety.org/10.1109/CLOUD.2010.50.

[45] Thönes, J. Microservices. *IEEE software.* IEEE. 2015, vol. 32, no. 1, p. 113–116. ISSN 1937-4194.

[46] Tsai, W., Bai, X. and Huang, Y. Software-as-a-service (SaaS): perspectives and challenges. *Science China Information Sciences.* Springer. 2014, vol. 57, no. 5, p. 1–15. ISSN 1869-1919.

[47] Yousafzai, A., Gani, A., Md. Noor, R., Sookhak, M., Talebian, H. et al. Cloud resource allocation schemes: review, taxonomy, and opportunities. *Knowledge and Information Systems.* february 2017, vol. 50, p. 350–364. DOI: 10.1007/s10115-016-0951-y. ISSN 0219-3116.

Appendix A

File structure

Structure of included files.

```
DIP
├── billing
├── FileManagement - Microservice source code
├── UserManagement - Microservice source code
├── deployment_dev - Kubernetes files for deployment
│   ├── billing
│   ├── filemanagement
│   ├── filemetadata
│   ├── kubernetes-ui
│   ├── metrics
│   ├── minio
│   ├── notifications
│   ├── postgres
│   ├── promentheus
│   ├── rabbitmq
│   ├── scripts -folder for scripts that were used during experiments
│   └── shared - shared configuration for microservices
├── FileMetadata - Microservice source code
└── Notifications - Microservice source code
```

Appendix B

Deployment manual

The demo application is available at https://github.com/MatejKolesar/DIP. Each microservice contains a README file with the needed steps for the creation of an image for the microservice. The folder `deployment_dev/shared` contains configurations for the shared microservices and all changes to the environment variables for the microservices come from that point. The concrete changes of the image versions are in the deployment files for each microservice.

To deploy each microservice from scratch you need to create the image first. Each microservice has a README which describes this process.

Prerequisites

- Kubectl
- Helm
- Optional - only needed if you want to compile from scratch:
  - Java 11 or higher
  - Spring Boot framework
  - Spring Data JPA

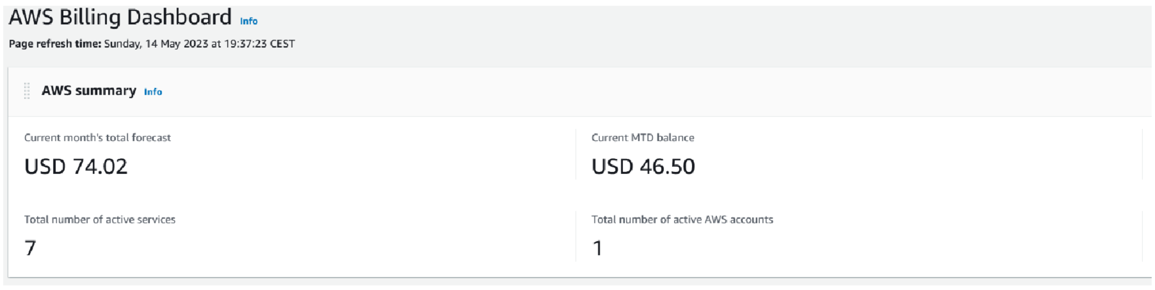
Deployment on local

This guide will walk you through the process of setting up a local environment. On a local environment, there is no security that is inherently present on AWS.

Prerequisites

Before you can deploy a Kubernetes application, you'll need to have the following prerequisites installed on your local machine:

- Docker
- Kubernetes CLI (kubectl)
- Minikube (or a similar Kubernetes distribution)
- A containerized application (Docker image) that you want to deploy


AWS Billing Dashboard Info

Page refresh time: Sunday, 14 May 2023 at 19:37:23 CEST

AWS summary Info

Current month's total forecast
USD 74.02

Current MTD balance
USD 46.50

Total number of active services
7

Total number of active AWS accounts
1

Figure B.1: AWS billing dashboard

Step 1: Start a Kubernetes Cluster

The first step to deploying a Kubernetes application locally is to start a Kubernetes cluster. To do this, you can use a tool like Minikube, which allows you to create a single-node Kubernetes cluster on your local machine. To start a new cluster with Minikube, run the following command in your terminal:

```
#minikube start
```

Step 2: Deploy the Sample Application

1. Clone the sample application repository from Github.
2. Open the Kubernetes deployment YAML file and update the variables with your AmazonMQ broker and S3 bucket information.
3. Ensure that you can access the EKS cluster from the terminal.
4. Deploy the sample application to your EKS cluster using kubectl apply.

Deployment on AWS

This guide will walk you through the process of setting up an Elastic Kubernetes Service (EKS) cluster with AmazonMQ and S3. Before trying on AWS be sure that you have an active AWS subscription as the free tier that they offer does not cover much and there is no warning for going over the free tier. For the current billing use the billing dashboard located under the profile tab. And also be sure to delete all created resources when they are not used as they run 24/7 and the bill increases.

Prerequisites

Before you begin, make sure you have the following:

- An AWS account
- The AWS CLI installed on your local machine
- The kubectl command-line tool installed
- A VPC configured in an availability zones (ideally in your region)

- **A security group or identity provider that allows traffic between the subnets**

Step 1: Create an EKS Cluster

1. Open the Amazon EKS console and click the „Create cluster“ button.
2. Choose a name for your cluster and select the Kubernetes version you want to use.
3. Under Networking, select the VPC and subnets you created earlier.
4. Choose the number of nodes and the instance type you want to use for your worker nodes.

Step 2: Create an AmazonMQ Broker

1. Open the AmazonMQ console and click „Create a broker.“
2. Choose the broker engine you want to use and the instance type you want to use for your broker nodes.
3. Configure the networking settings for your broker.
4. Choose a name for your broker and review your settings.
5. Create your broker.

Step 3: Create an S3 Bucket

1. Open the Amazon S3 console and click „Create bucket.“
2. Choose a globally unique name for your bucket.
3. Choose the region where you want to store your bucket.
4. Configure any additional settings you need for your bucket.
5. Create your bucket.

Step 4: Deploy and Configure the Sample Application

1. Clone the sample application repository from Github.
2. Open the Kubernetes deployment YAML file and update the variables with your AmazonMQ broker and S3 bucket information.
3. Ensure that you can access the EKS cluster from the terminal.
4. Deploy the sample application to your EKS cluster using `kubectl apply`.