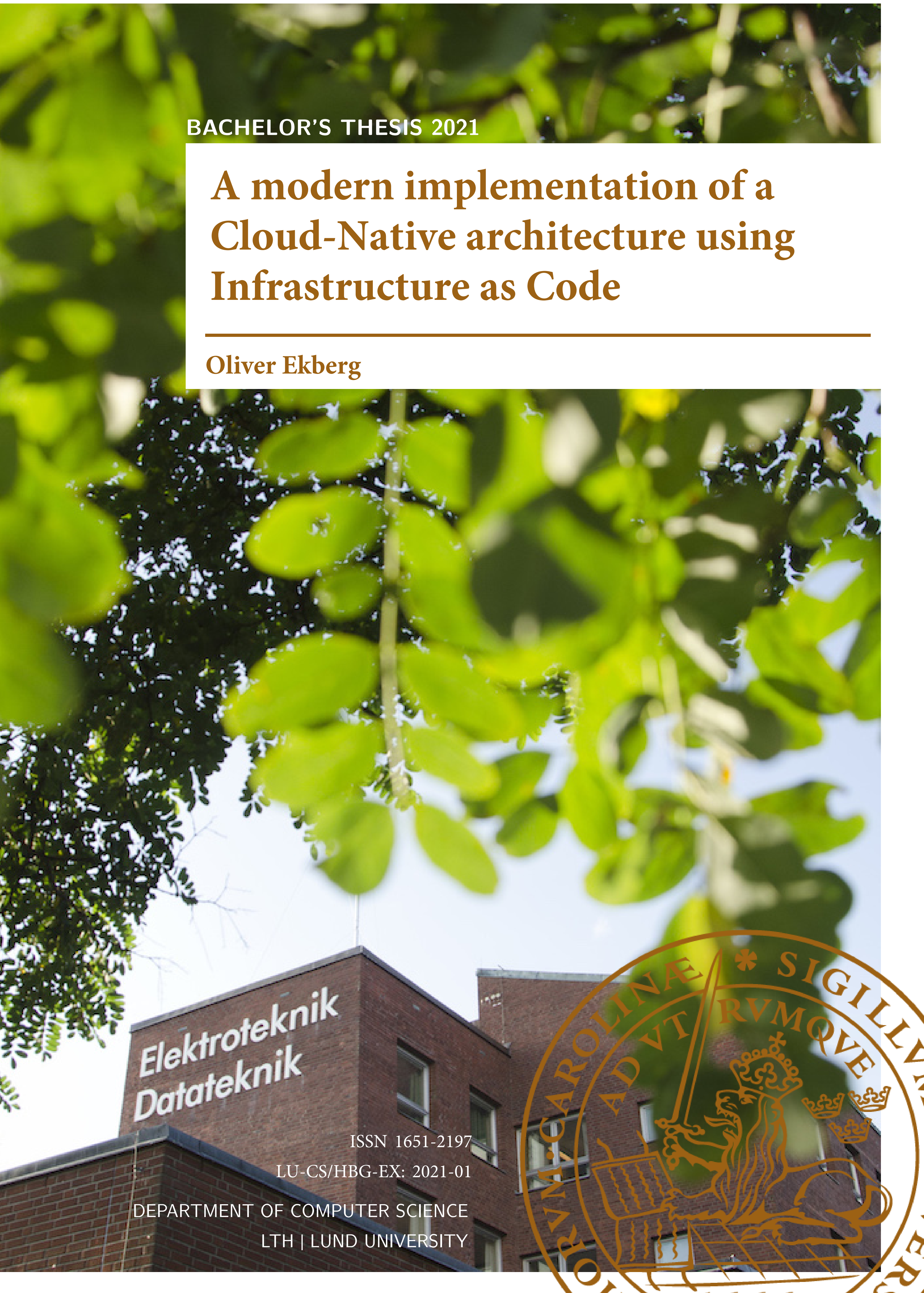
BACHELOR'S THESIS 2021

A modern implementation of a Cloud-Native architecture using Infrastructure as Code

Oliver Ekberg



ISSN 1651-2197
LU-CS/HBG-EX: 2021-01

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

A modern implementation of a Cloud-Native architecture using Infrastructure as Code

LU-CS/HBG-EX 2021-01

LUND UNIVERSITY
Campus Helsingborg

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:
Oliver Ekberg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Lunds universitet
Lund 2021

Abstract

With the rise of cloud service providers such as Amazon Web Services (AWS) and the tailored services they provide, companies using a traditional VPS-based architecture have to consider if they should move their architecture. Zmarta, a company providing financial comparison services in the Nordics, is considering moving its growing VPS- based architecture to a cloud-native one in AWS to allow for better scalability and reliability. This thesis work designs and implements a sample system and its infrastructure using modern techniques such as Event-Driven Architecture and Infrastructure as Code (IaC) to act as a proof of concept that can be used by Zmarta when they decide to make the change. Using interviews with Zmarta employees alongside literature studies, the topic of cloud engineering was explored, and relevant AWS services selected. Using the programming language JavaScript, both the infrastructure and applications that would run on it is programmed. The infrastructure is then evaluated based on measurements made on scalability and reliability. The results show how infrastructure could be written with a few lines of declarative code and how changes to that infrastructure and applications running on it could quickly and reliably be deployed. Also, it was shown that automatic scaling could be configured with virtually no upper limit. And, although not scientifically measured, it was discovered that the added cost of fully managed cloud services sometimes could be justified, when taking into consideration the cost of the developer hours needed to manage applications at scale.

Keywords: AWS, Cloud-Native, Infrastructure as Code, Event-Driven Architecture

Sammanfattning

Med ökningen av molntjänstleverantörer så som Amazon Web Services (AWS) och de skräddarsydda tjänsterna de tillhandahåller, måste företag som använder en traditionell VPS-baserad arkitektur överväga om de borde flytta sin arkitektur. Zmarta, ett företag som tillhandahåller finansiella jämförelsetjänster i Norden, överväger att flytta sin växande VPS-baserade arkitektur till en cloud-native arkitektur på AWS för att möjliggöra bättre skalbarhet och tillförlitlighet. Detta examensarbete utformar och implementerar ett provsystem och dess infrastruktur med moderna tekniker som Eventdriven arkitektur och Infrastruktur som kod, för att fungera som ett koncept som kan användas av Zmarta när de bestämmer sig för att göra förändringen. Med hjälp av litteraturstudier och intervjuer med anställda på Zmarta undersöktes ämnet molnteknik och relevanta AWS-tjänster valdes ut. Programmeringsspråket JavaScript användes för att programmera infrastrukturen och applikationerna som ska köras på den. Infrastrukturen utvärderas sedan baserat på mätningar gjorda på skalbarhet och tillförlitlighet. Resultaten visar hur infrastruktur kunde skrivas med några enstaka rader deklarativ kod och hur ändringar av infrastrukturen och dess applikationer enkelt och pålitligt kunde distribueras. Det visades också att automatisk skalning kunde konfigureras utan någon praktiskt taget övre gräns. Och även om det inte var vetenskapligt uppmätt upptäcktes det att den extra kostnaden för helt administrerade molntjänster ibland kunde motiveras när man tar hänsyn till kostnaden för de timmar som krävs av utvecklare för att hantera applikationer vid stor skala.

Nyckelord: AWS, Cloud-Native, Infrastruktur som kod, Eventdriven arkitektur

Foreword

This is my thesis work for my bachelor's degree in computer science and software engineering at Lunds Tekniska Högskola (LTH) during spring 2021.

Acknowledgments

First and foremost, I would like to thank Zmarta for letting me perform my thesis in cooperation with them. By providing a real and challenging problem, I was able to find and learn about a highly relevant topic at the time of writing this. I am especially thankful for the cooperation and guidance received from Zmarta's architect and my supervisor, Jonas Brauer, and their head of IT operations, Erik Holmqvist.

I am also in significant debt of gratitude to my supervisor and mentor at LTH, Christian Nyberg. With his thesis experience, he was able to guide me through the process of writing this rapport. Suffice to say that this rapport would not be of the same quality without his help.

During the final stages I got valuable feedback on my rapport, which resulted in changes that led to greater comprehension. For this I want to thank my examiner at LTH, Christin Lindholm.

Lastly, I want to thank my fellow coursemates André Reis and Sebastian Nordqvist, for all the help and valuable discussions we had in the early stages; they really helped me get going when I was stuck.

List of contents

1 Introduction

This document outlines a bachelor's thesis, performed in cooperation with the company Zmarta Group AB. It means to explore what a Cloud-Native architecture is, and how it can be implemented using Infrastructure as Code (IaC) to fulfil Zmarta's current needs.

1.1 Background

When designing a tech organization's architecture, there are usually two viable options: cloud-hosted and cloud-native. Using cloud-hosted, the organizations pay a cloud computing provider for one or many virtual private servers (VPSs) where they orchestrate their software. This is, however, in many cases, not the most efficient architecture nowadays, with well-established cloud-computing providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure providing native cloud services, tailored for their specific areas.

One disadvantage of managing a VPS-based architecture is keeping up with security. A company's operations team must ensure that the operating system and any software running on the architecture are up-to-date and continuously patched for security loopholes. They also need to set up and manage a system for controlling users' permissions that might be needed to access the servers, such as software developers tweaking their applications.

Another disadvantage to this approach is scaling and reliability. For software companies, it is crucial to keep up with demand; a website cannot simply become slower, or in the worst-case crash, just because the number of active users unexpectedly spiked. Things like this can cost companies a lot of money as well as the trust of their customers. Cost is another reason to consider other approaches. Hosting servers in the traditional sense requires the company to pay a flat rate for hardware, no matter the actual usage. This kind of ties into the scaling aspect, as it is crucial to have enough power to cope with the highest expected traffic. However, all this hardware will most likely sit idle most of the time, still costing the same, even though it is not utilized. It is also important to consider the cost in terms of person-hours needed to manage such systems.

The time it takes to set up and maintain infrastructure for running software is yet another critical factor. Having a VPS-based architecture often means that a team dedicated to doing just that, often called the operations team, is required to facilitate an effective environment for developers to run their software. This is inefficient as developers are highly dependent on the operations team for many things, especially when a new solution is required.

To solve these issues mentioned above, a cloud-native architecture can be implemented. It allows smaller companies to utilize the fact that big cloud providers have well-developed, user-friendly systems. Also, since they operate at scale, they can offer auto-scaling and pay-as-you-go cost models.

This thesis work will investigate a more traditional architecture and attempt to move it into the cloud. In cooperation with Zmarta Group AB, a small system will be set up on both their current architecture and the cloud to compare speed and cost. Zmarta is a software company specializing in comparison services in the Nordics for things like consumer loans, mortgage, and insurance, to mention a few. Zmarta is in a phase where they are considering to move their current applications into the cloud, and this thesis will serve as a proof of concept for just that.

1.2 Purpose

The purpose of this thesis is to investigate how a more traditional VPS-based architecture can be transformed into a hosted cloud-native one and the benefits of using a cloud solution in terms of response time, throughput, reliability, and cost. It is expected that the cloud will be more challenging to comprehend in the beginning, but then save time and resources, especially at scale.

1.3 Goals

This thesis will delve into what makes cloud architecture advantageous, how it is commonly used, and its drawbacks. After investigation, a small system with realistic functions will be set up in Zmarta's current architecture and the cloud. The cloud solution will, in part, implement an event-based architecture to align with Zmarta's current priorities. AWS will be the cloud platform used for this thesis, chosen by Zmarta as they currently use some of AWS's services. These systems will then be evaluated based on findings and measurements.

1.4 Problem description

The following questions will be discussed in the thesis and needed to fulfill the whole project.

1. How can infrastructure be provisioned in the cloud in an efficient and traceable manner?
2. How can the release of application-level changes be automated?
3. What is the difference between different levels of managed services in terms of cost and features?
4. How can autoscaling be configured and utilized to adapt for a variable load?
5. Which AWS services are appropriate for fulfilling the company's needs?
6. What will the company gain by adopting a cloud-native infrastructure on AWS?
7. What will the company lose by adopting a cloud-native infrastructure on AWS?
8. What comparisons and measurements should be performed to evaluate the different architectures?

1.5 Motivation

This thesis was chosen to broaden the knowledge in the application hosting world. Also, as it is increasingly common for companies to run their stack in the cloud, it would not be surprising if every software developer needs to work in the cloud as part of their everyday work. Therefore, knowledge in this area might lead to future work prospects.

Using cloud computation services also has positive effects on humanity as a whole, which further motivates me to do this research. Not only can we be more efficient with the use of hardware, thus helping the environment, but also in terms of excellent reliability and availability. Some critical systems such as hospital medical journals are entirely dependent on this, and moving them to the cloud might be a way to a more secure, digital future.

The thesis is also highly relevant for Zmarta as they currently want to move their system into the cloud. If everything goes as intended, Zmarta will kick start this journey using my findings, which undoubtedly saves them time and money.

1.6 Limitations

In order to make the thesis fit the timeframe, some limitations need to be put in place:

- AWS is a giant cloud provider, letting its users pick from over 200 different services [1]. This thesis will only look into services and concepts relevant to Zmarta.
- Only a small system will be set up. In order to preserve time, no actual business logic will be built. This is not perfect but will mimic a real-world scenario.
- The small system will be backend only, meaning that no user-facing application will be developed.
- The testing in Zmarta's current architecture might be limited as we do not want to disrupt anything currently running there. Therefore, only measurements allowed by the operations team will be performed.
- All cloud prototyping will be done in AWS, as is the cloud provider of choice for Zmarta. Names of services will differ between AWS and other providers; however, their concepts still apply.
- Some of the more expensive services might not be tested.
- Administration of user rights and other surrounding best practices will not be considered as it is a broad subject that will not fit into this thesis.

2 Technical background

This chapter aims to give a better understanding of core concepts needed for cloud computing, relevant AWS services and introduces the topic of containerization. On top of that, Zmarta's current architecture and technical infrastructure are introduced and dissected.

2.1 Cloud Service Levels

Cloud vendors such as AWS offer services at different levels to fit their customers' needs. Knowing these can help a company to better understand what it is actually paying for.

2.1.1 Infrastructure as a Service (IaaS)

Infrastructure as a Service is the lowest service level cloud providers typically offer. The vendor takes care of hosting the physical machines and making sure they are highly available, while the consumer is responsible for everything else like operating system and application servers. Products offered at this level are compute instances and storage buckets [2].

2.1.2 Platform as a Service (PaaS)

Platform as a Service is further abstracted compared to IaaS. On top of handling everything IaaS does, PaaS also takes care of operating systems and other software needed to run and scale applications. This means that developers can focus on developing applications, and the cloud provider ensures that these applications are highly available and scale according to their needs. Examples of products offered at this level are managed container orchestration services and managed databases [3].

2.1.3 Serverless

Serverless is yet another abstraction. Instead of scaling the number of instances to match the load, Serverless scales on a per-request-basis. That means that this type of service is more suitable to adapt to sudden changes in load and scales up virtually infinite and down to zero. Due to their nature, Serverless services are usually event-driven functions that get triggered by things like messages and HTTP requests [4].

2.1.4 Software as a Service (SaaS)

Software as a Service is the highest level of abstraction. The consumer can simply use a finished piece of software without having to do any sort of configuration themselves. Examples of SaaS range from commonly used software as search engines to more advanced business-oriented solutions like customer relationship management systems (CRMs) [5].

2.2 Containerization

Containerization is a concept in application development that challenges the traditional use of virtual machines (VMs). Instead of running VMs with their own operating system on a hypervisor, containerization runs containers on a container engine. The advantage of doing this is that the overhead of running individual OSes is eliminated and taken care of by the engine [6].

One feature of containers is that they are portable by design. This means that a container developed on one person's computer will run in exactly the same way on any system using the same container engine.

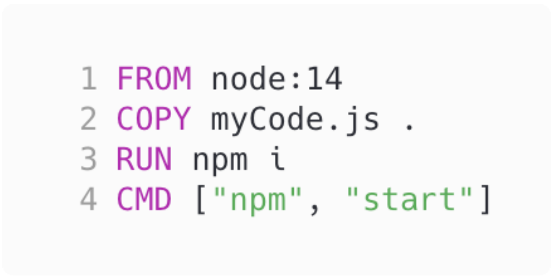
The process of creating containers that can be run on a container engine consists of three core steps: defining the container, creating an image from the definition, and then running the image as an actual container [7].

2.2.1 Docker

Docker is an open-source container runtime that has become the standard for many managed services such as AWS ECS (2.4.5) and AWS Fargate (2.4.6) [8].

2.2.1.1 Dockerfile

When defining an image within Docker, something called a Dockerfile is used. The Dockerfile is a set of instructions that creates the described image when the docker run command is executed [9]. Figure 1 shows a simple Dockerfile that creates a Node.JS application.



```
FROM node:14
COPY myCode.js .
RUN npm i
CMD ["npm", "start"]
```

Figure 1. Simple Dockerfile

2.2.1.2 Docker Image

A Docker image is the result of building a Dockerfile. It is a description of how the resulting container will function when run. Images can be built upon other images, which is shown in Figure 1 with the FROM-command [10]. They can also be packaged with dependencies installed and binaries already built, which means that this process is repeated once, even if the image is instantiated into multiple containers.

2.2.1.3 Docker Container

A Docker container is the runnable version of an image. It is however possible to alter the container's behavior with configuration parameters given when the container is created. When a container is destroyed, any data created or modified within that container is deleted too. This can be mitigated by providing something called volumes, which can be mapped to persistent storage [10]. Although it is possible to use volumes with managed services such as ECS (2.4.5), it is impossible with a fully serverless option like Fargate (2.4.6) since the machines running the containers are completely abstracted.

2.2.1.4 Docker health check

For Docker to know if a given container is operational or not, something called a health-check can be implemented. This is useful because Docker can restart or create new containers if one is found out to be unhealthy. An application-specific health check can be built and then exposed through the container in a standardized way using the HEALTHCHECK instruction in a Dockerfile [9]. Managed container orchestration tools such as AWS ECS (2.4.5) can use this status to send alarms and even take automatic action.

2.2.1.5 Docker Registry

Docker registry is an application for storing and distributing Docker images [11]. The application can be downloaded and ran on a self-hosted service or bought as software as a service (SaaS). Docker provides its own managed registry called Docker Hub [12]. AWS also provides a container registry called Elastic Container Registry (2.4.13).

2.3 Zmarta's Current Setup

To facilitate development in autonomous teams, Zmarta has gone from a monolithic to a microservices architecture. This allows multiple teams to work with different stacks without affecting each other. To achieve this without limiting the teams, all applications are containerized using Docker.

2.3.1.1 Monolith Architecture

Monolith architecture is where multiple different pieces of logic are combined into one program. These are typically easy to develop and maintain at a low scale but become unfeasible when growing too big [13].

2.3.1.2 Microservice Architecture

To keep up with today's application needs, it is common to build distributed systems. This is where the Microservices architecture comes in. Achieving this architecture is done by splitting different part of an application into logical parts, microservices, that follows the single responsibility principle (SRP). While being more complex to develop and run than a monolith, microservices scales better in terms of throughput and the number of developers that can work on them [13].

2.3.2 Application Hosting

Zmarta uses two EC2 (2.4.4) instances to run their production applications. Traffic is load-balanced between the two instances using Nginx [14], based on the instance's health. However, Nginx will not take individual containers' health into account, meaning that it will still get routed traffic if one container is unhealthy. So, there is redundancy in terms of infrastructure, but not on a software level.

2.3.3 Container Orchestration

To orchestrate its containers running on the two EC2 instances, Zmarta has implemented the orchestration tool Ansible [15] on a third EC2 instance. On the deployment of a service, Ansible is responsible for updating the Nginx configuration associated with the service, turning off the old containers, and starting the new and injecting secrets, which are sensitive parameters such as API credentials, into the new containers.

To avoid duplication and streamline deployments, Zmarta's operations team has set up default playbooks in Ansible for deploying microservices in the form of Docker containers.

2.3.4 Development Workflow

At Zmarta, software developers push their code to repositories on GitLab. For each application, there is a repository containing application logic and a Dockerfile describing how the service's image will be built.

When pushing code to the repository, a GitLab CI/CD pipeline is created and runs various quality controls, builds the image, uploads it to a container registry, and then triggers an Ansible playbook for deployment. A set of EC2 instances are provided as the computation power behind the pipelines.

2.4 Amazon Web Services (AWS)

This section is the result of Phase 2, which is explained in greater detail in section 3.2. It explains the concept of Infrastructure as Code and introduces AWS services that potentially could be useful to fulfil Zmarta's needs.

2.4.1 Infrastructure as Code (IaC)

With AWS, it is possible to provision and manage all services through Amazon's website in a graphical interface. However, it is also possible to do the same with code. This is where Infrastructure as Code (IaC) comes in. With IaC, developers can specify the infrastructure in some standard format such as YAML or even in modern programming languages such as TypeScript. One benefit of this is that developers get to work with a format that they already know and leverage assistance from their integrated development environment (IDE) of choice [16].

Since the infrastructure is defined in text documents, it is simple to version-control it with Git [17]. This will allow for full traceability, which can be helpful when debugging issues. Like regular code, the infrastructure can now be put through best practices such as code reviews (CR). By having others review changes to the infrastructure definition before it gets deployed, potentially catastrophic outcomes may be prevented.

Speed and scalability is another reason to use IaC. Many companies, including Zmarta, leverage different application environments to enable testing in a production-like environment without risking any unwanted consequences to the actual production environment. If one wanted to set up a new environment through a GUI, it would be very difficult to configure it precisely the same way as the production one, even with all the steps written down. And not only that, when the infrastructure changes, the changes have to be reflected in all environments. With IaC, the template used for production can create multiple identical environments with only slight changes to the definition itself. The same goes for geographical expansion when a company wants to provision its infrastructure in a different geographical location.

2.4.2 AWS CloudFormation

CloudFormation is AWS's answer to IaC. CloudFormation lets you define your desired infrastructure in either YAML or JSON, which then gets uploaded and run on AWS to release it. As mentioned above, it makes it possible to provision infrastructure with a single command and deleting it with another. The option to delete the whole stack is helpful since it ensures that all services are taken down, therefore not costing companies any more money.

When a CloudFormation file is run, AWS compares the current state of the infrastructure to the desired and takes action to ensure these changes. Therefore, it is recommended not to make changes with CloudFormation and the GUI, as this will most likely cause synchronization issues [18].

2.4.3 AWS Cloud Development Kit (CDK)

More recently, Amazon released a new product for IaC - CDK. CDK serves the same purpose as CloudFormation, and in fact, compiles to CloudFormation. The difference lies in that CDK allows you to define the infrastructure using programming languages such as TypeScript and Java instead of static configuration files such as YAML and JSON. This means that traditional programming paradigms can be used. For example, if one wanted to link service A with service B in a static file, that would mean having to copy some identifier from one service to the other. With an object-oriented programming language such as TypeScript, it is instead possible to pass a reference of one service object to another, which is more readable and safer due to type checking. Even loops and conditional statements can be used [16].

2.4.4 AWS Elastic Compute Cloud (EC2)

EC2 is a service for provisioning highly available VPSs within AWS. It has support for nearly 400 different instance types, which are optimized and sized for different needs. EC2 follows a pay-as-you model, which means the number and size of the instances can be adjusted based on the load and consequently reduce the cost when the load is lower.

EC2 is IaaS, which means that AWS takes care of hardware, networking, and storage, and the user is responsible for the operating system, runtime, and application [19].

2.4.5 AWS Elastic Container Service (ECS)

ECS is a managed container orchestration service that can be installed onto existing EC2 instances. ECS ensures that the specified number of containers run on the EC2 instances and automatically scales them based on load. However, since it is an add-on to EC2, it cannot scale the hardware, which means users still have to scale their EC2 instances independently from their containers [20].

2.4.6 AWS Fargate

Fargate is a subset of ECS, with the difference that it is serverless. This means that a user does not need to scale and manage EC2 instances. Fargate does not allow containers to be stateful, which is possible with ECS since the user provides their EC2 instances.

With Fargate, users only provide their containers, and AWS ensures these containers are run and scaled according to the configuration. However, it is up to the user to ensure the containers themselves are updated and works as intended [21].

2.4.7 AWS Lambda

Lambda is AWS's answer to a serverless function. That means that users have to supply the code to run and nothing more. With this service, users pay per function invocation based on how many milliseconds the function runs. When no invocations occur, the user will not be charged [22].

2.4.8 AWS Simple Queue Service (SQS)

SQS is a fully managed messaging queue. Its primary purpose is to decouple systems by providing a set interface for producers and consumers to communicate. Producers put messages on the queue, and consumers poll the queue for updates. When a message is processed, the consumer removes it, which means that a message is only delivered to one consumer.

There are two different types of queues available in SQS: standard and FIFO. Standard is cheaper and supports nearly unlimited throughput. It guarantees that a message gets delivered, but it does not guarantee that it does not get delivered more than once nor the order. On the other hand, FIFO guarantees that the order is maintained and that a message is only delivered once. However, it costs more and has limited throughput [23].

2.4.9 AWS Simple Notification Service (SNS)

SNS is like SQS also a fully managed messaging service. With it, the user defines topics that products publish messages to and consumers subscribe to. SNS also provides a standard version and a FIFO version. These versions have the same attributes as the ones described for SQS, with the addition that the standard supports more topics and subscribers than the FIFO version [24].

2.4.10 AWS Simple Storage Service (S3)

S3 is an object storage service for storing any amount of data with high durability. It is designed for 99.999999999% durability. The pricing is monthly and depends on how much data is stored and the number of reads and writes.

Objects in S3 are organized into different user defined buckets. A bucket acts like a regular file folder and allows for logical grouping of objects, however, it is not possible to have nested buckets. With a bucket it is possible to configure permissions which then applies to all its containing objects.

In addition to accessing S3 through other AWS services such as Lambda, it is also possible through the HTTP API. This means that images can be hosted there and then directly accessed by website visitors through the associated HTTP URL [25].

2.4.11 AWS Relational Database Service (RDS)

Although relational databases such as MySQL and PostgreSQL can be installed on an EC2 instance, AWS provides RDS, a managed relational database service. RDS supports industry-standard engines such as MySQL and PostgreSQL, to name a few, and their own engine, called Aurora.

While the traditional engines have to be scaled by changing the underlying instance type, Aurora is fully managed in that it scales automatically. Moreover, since Aurora's cost is only based on the storage and I/O used, it is impossible to over-provision it.

With RDS, it is possible to take automated snapshots and backups, which can be stored directly on S3. Consequently, a database can be restored to a previous backup/snapshot by simply choosing it from the S3 bucket [26].

2.4.12 AWS DynamoDB
DynamoDB is a managed NoSQL database that promises to deliver fast performance at any scale. It is serverless and scales automatically to handle spikes in load while minimizing cost.

Just like RDS, DynamoDB offers backups, but it also offers something called point-in-time recovery. This means that a collection can be restored to the state it had any second within the last 35 days [27].

2.4.13 AWS Elastic Container Registry (ECR)
To store Docker images used by, for example, ECS and Fargate, AWS provides their own, fully managed registry - ECR. It utilizes S3 for storage, which provides high availability and durability [28].

2.4.14 AWS Application Load Balancer (ALB)
When dealing with multiple instances of a server, load balancing is needed to distribute the load between them. For HTTP traffic AWS offers a specialized load balancer called Application Load Balancer (ALB). It can route traffic to EC2 instances, containers, IP addresses, and Lambdas. ALB has exceptional ECS support because container ports can be specified dynamically and then registered in the load balancer by ECS [29].

2.4.15 AWS Secrets Manager
For managing secrets and configuration AWS provides something called Secrets Manager. Secrets Manager is a fully managed service that ensures secrets are stored and communicated safely. It also has a feature allowing for automatic secret rotation.

It is possible to securely access secrets with both AWS SDK and CloudFormation. This means that secrets can be injected into applications when the infrastructure is provisioned using CloudFormation and fetched runtime on the application level through the SDK [30].

2.4.16 AWS Virtual Private Cloud (VPC)
To communicate sensitive information between AWS services, such as retrieving secrets from Secrets Manager, AWS offers a virtual private cloud. AWS VPC creates logically isolated networks that can be configured to be either public or private. As such, backend services that end-users should never be allowed to access, such as databases, can be protected within a private network only accessible by whitelisted services [31].

2.4.17 AWS CloudWatch
CloudWatch is a service that allows developers to observe their whole infrastructure and applications running on AWS. It can be used to collect and visualize logs and metrics, to mention a few. These data points can then be used to trigger alarms and automatically scale EC2 instances and ECS clusters [32].

3 Methodology and Analysis

This chapter describes the four different phases of the thesis work and how they were carried out. Each phase is presented in its on section. The phases are:

- Phase 1: Collecting information
- Phase 2: Application and infrastructure design
- Phase 3: Implementation
- Phase 4: Evaluation

As shown in Figure 2, Phases 1 and 2 were performed iteratively. This was due to the need for more information during the design. However, when the design was complete, Phases 3 and 4 could be carried out sequentially.

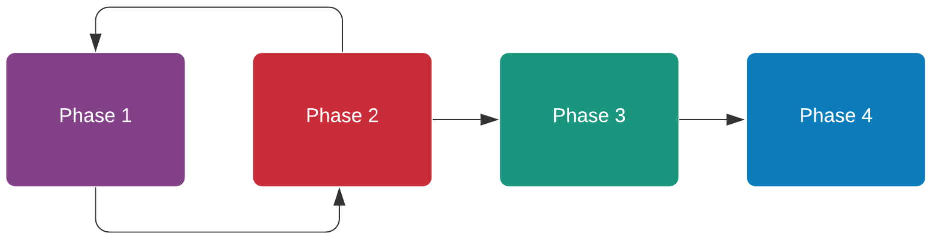


Figure 2. Flow of phases

3.1 Phase 1: Collecting information

Phase 1 consisted of information gathering in terms of interviews and literature studies to determine what AWS services could help fulfill Zmarta's needs. All interviews were conducted in a semi-structured fashion by preparing a few questions and letting the conversations flow, following Soren Lauesen's recommendations in his book "Software Requirements: Styles and Techniques" [33]. Using this interview type was especially useful as very little was previously known about Zmarta, their setup and cloud engineering.

All interviews were held as remote video conferences, due to the global covid-19 pandemic taking place at the time. Notes of key areas were written down during the interviews using pen and paper. After that, the design work continued based on the answers, and if needed, more literature studies were performed to get a deeper understanding of the discussed topics and services.

This section describes how information was collected but does not go into deeper details about the information itself. Instead, this is done in the chapter *Technical background*, which is a direct result of Phase 1.

3.1.1 Getting to know Zmarta and its needs

The first introduction to Zmarta's systems was given by Erik Holmqvist, the company's head of IT operations. Five questions, shown in Table 1, were prepared beforehand, and the interview took 75 minutes due to deep discussions regarding Infrastructure as Code in particular.

During the interview, Erik explained how they provision and maintain their microservices on a higher level. He also mentioned a few deficiencies in their current setup that they would like to improve. As Erik is very familiar with AWS, he also gave some pointers on what to do and what not to do in the cloud. He recommended that VPC and CloudWatch should be configured from the beginning, as this could be tedious to configure later on.

Table 1. Interview questions for Erik

Identifier	**Question**
1.1	What infrastructure is currently used to host Zmarta's applications?
1.2	How are changes to that infrastructure made?
1.3	Are there any deficiencies to Zmarta's current setup?
1.4	How is variable load handled?
1.5	How are recurring tasks scheduled?

Another interview was conducted with Zmarta's software architect, Jonas Brauer. For this interview three questions, shown in Table 2, were prepared beforehand, and the interview took 25 minutes. Throughout the interview, Jonas explained Zmarta's architecture and how they develop microservices in Node.js, a runtime for JavaScript backend applications [34]. He also recommended using AWS Secrets Manager to manage secrets such as database credentials and API keys.

When asking the question “What kind of software architecture is used?”, Jonas described that Zmarta previously used a monolith architecture, but now uses a microservice architecture. He also mentioned that they are starting to incorporate an event driven architecture to allow for further separation of concerns in applications. As this type of architecture will be used even more in the future at Zmarta, Jonas recommended that services allowing for this type of asynchronous messages, or events, should be investigated.

Table 2. Interview questions for Jonas

Identifier	**Question**
2.1	What programming languages are used?
2.2	What kind of software architecture is used?
2.3	How is code delivered into production?

3.1.2 Exploring AWS concepts and services

After getting an introduction to Zmarta's current architecture, it was time to get to know the AWS ecosystem and collect a list of services that could be used to implement a corresponding architecture natively in the cloud. At first, to get a higher-level understanding of AWS, online recordings of official AWS events were consumed through YouTube [16]. Later each potential service was reviewed by reading its official documentation on the AWS website. The final list of services is shown in 2.4.

3.2 Phase 2: Application and infrastructure design

After collecting information on Zmarta and its current architecture, and AWS services that could potentially be used, it was time to design an architecture that could be built in the cloud. In order to design a relevant architecture according to best practices, literature studies were made. Also, AWS's interviews in the series "This Is My Architecture" [35] were consumed to get inspiration from other companies' implementations. The final infrastructure design is shown in Figure 4.

3.2.1 The application

Before designing infrastructure, the application that was going to run on the infrastructure had to be invented. The application did not need to implement any sound logic; it just had to illustrate patterns commonly used by Zmarta. As shown in Figure 3, the result was an HTTP application that both perform actions in real-time and asynchronously. Also, a recurring task was added.

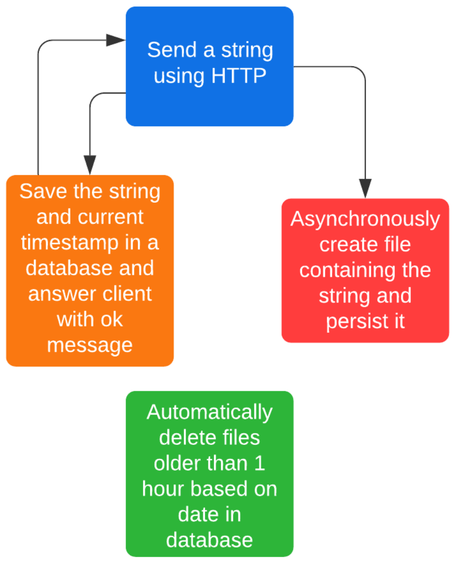

Send a string using HTTP
Save the string and current timestamp in a database and answer client with ok message
Asynchronously create file containing the string and persist it
Automatically delete files older than 1 hour based on date in database

Figure 3. Application design

3.2.2 The infrastructure

With the application invented, the infrastructure on which the application should run on had to be designed. Suitable AWS services were chosen based on the information gathered in Phase 1. The end result of what services were chosen and how they relate to each other is shown in Figure 4.

3.2.2.1 Networking

Although networking was not initially a part of the thesis, some core fundamentals had to be studied to ensure that the result was not inherently unsafe. This was further reinforced by the fact that Erik mentioned the importance of this to Zmarta in Phase 1, as they work with sensitive data.

A virtual private cloud (VPC) was used as a base for the architecture to secure communication between services. The VPC was split into two subnets, one public and one private, where the public subnet would contain all services that needed contact with the internet. All other services would be placed in the private subnet to allow security as default [36].

3.2.2.2 Handling real-time HTTP traffic
For handling incoming HTTP requests, a load balancer was placed in the public subnet. An ALB was chosen since it is recommended by AWS when working with elastic container service (ECS) [37].

The ALB was then connected to an ECS cluster containing multiple HTTP service instances running inside Fargate. Three instances were chosen as the minimum number to be running at all times.

3.2.2.3 Handling asynchronous side-effects
To ensure business-critical side-effects that can be processed in the background, a queue which can contain messages and a consumer that processes the queue was added [38]. For this thesis, SQS was chosen for the queue and Lambda for the consumer. An event-driven architecture was adopted using SNS as the messaging bus to put tasks on the queue in a decoupled manner. More consumers could be added to this bus, but for this thesis one was enough to illustrate the point.

3.2.2.4 Handling recurring tasks
Batch tasks that run on a regular schedule can be important for compliance and application-specific rules and are heavily used by Zmarta. To implement such behavior on AWS, Lambda was chosen as it is cost-effective and only runs when necessary [39].

3.2.2.5 Handling persistent data
As there was a need to persist files, AWS's go-to object storage S3 was chosen. For the real-time data, the choice was between RDS and DynamoDB. In the end, DynamoDB was chosen for its high performance and serverless nature.

3.2.2.6 Logs
Logging is a crucial part of application development to allow tracing and debugging. In AWS, CloudWatch is used for this purpose. CloudWatch was therefore connected to the HTTP service, the queue consumer, and the background job.

3.2.2.7 Secrets
For this application to function, it was critical that secrets could be kept safe and highly available. Due to this, and the fact that Zmarta already used it, AWS Secrets Manager was chosen.

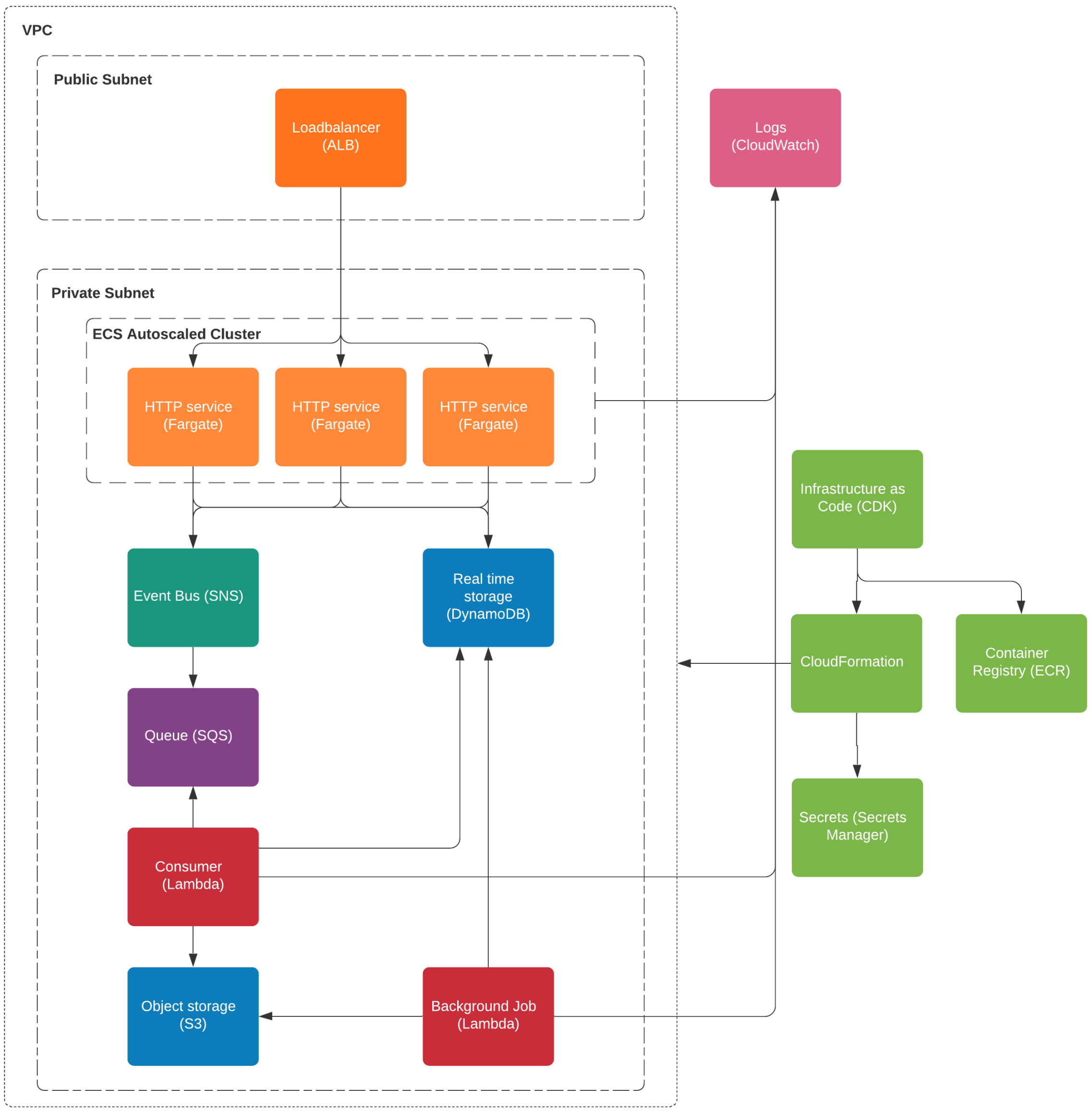


Figure 4. Infrastructure design

3.3 Phase 3: Implementation

This section describes the phase in which both the application and the infrastructure were implemented. Please note that all commands were executed on a Mac running macOS 11.2.1 and might be different in other operating systems.

During this phase, the agile development methodology Kanban [40] was used. It was chosen because the topic of IaC was unknown and could not be planned in sufficient detail ahead of time. Another reason for choosing Kanban was the limitations it puts on the number of concurrent tasks. By setting two tasks as the limit, greater focus could be achieved. A minimalistic Kanban board with the states “To do”, “In progress” and “Done” was created using the free online tool Trello [41]. Before starting the implementation, the work was split into small tasks which then were placed in the “To do” column. These tasks were then moved into “In progress” when they were being worked on, and finally to “Done” when they were finished.

3.3.1 Prerequisites

This section describes programs and tools that were installed and configured to develop locally. It also explains some commands that will be used throughout the phase.

3.3.1.1 Installing dependencies

Since all code was written in JavaScript, the Node.JS runtime and node package manager (NPM) were required. These were installed by downloading the appropriate installer and running it [42]. Docker Desktop was also installed to allow building images and running containers locally [43].

To communicate with AWS, their command-line interface (CLI) was downloaded and installed [44]. During the installation, an access key and access secret were generated by logging in to the AWS console and following AWS's official guide [45]. These credentials were then used to configure the AWS CLI by running *aws configure*.

Now that the AWS CLI was installed and connected to the AWS account, the CDK CLI could be installed by issuing *npm install -g aws-cdk*.

3.3.1.2 Commonly used commands

Commonly used commands are shown in Table 3.

Table 3. Commonly used commands

Command	**Description**
mkdir my-app	Creates a new folder called *my-app* in the current directory
cd my-app	Changes the current directory to *my-app*
npm init -y	Creates a new npm project with default options
npm install package-name	Installs the package *package-name*
node myfile.js	Runs myfile.js using the Node.JS runtime engine
docker build -t api .	Builds an image with the name *api*, based on a Dockerfile in the current directory
docker run -p 80:45 api	Runs the image *api* and maps port 80 in the container to port 45 on the host system
cdk init app –language javascript	Creates a new cdk application structure from a bare-bones template
cdk deploy	Deploys the application in the current directory
cdk destroy	Removes all provisioned resources associated with the application in the current directory

3.3.2 HTTP service

The HTTP service responsible for handling incoming requests was the first to be implemented. A list of requirements was created to ensure that the service implemented all desired features. This list is shown in Table 4.

Table 4. Requirements for HTTP service

Unique identifier	Requirement
1.1	The service should implement code to slow down requests artificially
1.2	The service should accept the parameter 'name' in the query
1.3	The service should generate a UUID for the new entity
1.4	The service should publish an event on the event bus
1.5	The service should create a new document in the database
1.6	The service should respond with a confirmation message and a 200-status code if everything went as expected
1.7	The service should respond a with a generic error message and a 500-status code if something went wrong
1.8	The service should log any thrown error to STDERR
1.9	The service should implement a Dockerfile for containerization

First, a new npm project was created in a new directory. To enable the desired functionality, three dependencies were then installed: *express, aws-sdk,* and *uuidv4*. Express is a popular framework used to handle HTTP requests [46]. Express was selected for its simplistic nature. The AWS SDK was chosen as it is the official way to communicate with AWS programmatically. UUIDv4 was chosen for id generation. A UUID was chosen instead of a traditional sequential ID since it has many possible combinations, which allows for a client-side generation with a negligible risk of collisions with existing entities. These specific dependencies were also chosen because they all are used in Zmarta's existing Node.JS services.

A file called server.js was created to contain all code needed to fulfill the requirements. In more important and realistic applications, one would not put all code in one file but instead create a scalable directory structure to allow for small, logically structured components. Nevertheless, for this thesis, one file was chosen to showcase the code more easily. The resulting code in server.js is shown in Figure 5.

```
 1 const express = require('express')
 2 const os = require('os')
 3 const AWS = require('aws-sdk')
 4 const { uuid } = require('uuidv4')
 5
 6 AWS.config.update({ region: 'eu-west-1' })
 7
 8 const app = express()
 9 const dbClient = new AWS.DynamoDB.DocumentClient()
10 const snsClient = new AWS.SNS()
11
12 // Introduce artificial slowdown to simulate real business logic
13 app.use((req, res, next) => {
14   for (let i = 0; i < 1e9; i++) {}
15   next()
16 })
17
18 app.get('/create', async (req, res) => {
19   try {
20     const name = req.query.name
21     const id = uuid()
22
23     await snsClient.publish({
24       Message: id,
25       TopicArn: process.env.SNS_ARN,
26       MessageGroupId: 'NAME_GROUP'
27     }).promise()
28
29     await dbClient.put({
30       TableName: process.env.TABLE_NAME,
31       Item: {
32         id,
33         name,
34         createdAt: Date.now(),
35         processedAt: null,
36         deletedAt: null,
37         stage: 'created'
38       }
39     }).promise()
40
41     res.status(200).json({ message: `Successful created on host: ${os.hostname()}` })
42   } catch (err) {
43     console.log(err)
44     res.status(500).json({ message: 'Internal server error, check logs for details' })
45   }
46 })
47
48 app.listen(process.env.PORT, () => { console.log(`App listening on port ${process.env.PORT}`) })
```

Figure 5. HTTP service code

On lines one through ten, all needed dependencies were included and initialized. The AWS SDK was configured to region *eu-west-1*, which is the chosen region for the entire thesis. To fulfill requirement 1.1, an Express middleware [47] was introduced. This middleware was configured to run when receiving any HTTP request and would perform one billion iterations to create the artificial load. This loop could have been placed with the actual HTTP handler, but a middleware was chosen instead as it allows for more straightforward code.

After that, it was time to implement the handler for an HTTP endpoint that would become the entry point to this service. To allow for simple testing in a browser, a GET listener on the path */create* was created. Inside it, requirements 1.2 through 1.8 were implemented. Some properties such as the SNS topic, DynamoDB table name, and the listener port could be changed based on the infrastructure. Therefore environment variables were used to allow CDK to inject them during a deployment dynamically.

To run the HTTP service on Fargate, it had to be containerized, a topic explained in 2.2. A Dockerfile was created in the same directory as server.js and contained the code shown in Figure 6. The hashtag symbol (#) denotes comments in the Dockerfile, which clarifies the different steps. To verify syntactical errors, the image was built locally. However, it was not possible to verify the actual functionality of the service locally as it would require the AWS infrastructure, which had yet to be implemented.

```dockerfile
# Base image upon the latest supported version on Node
FROM node:14

# Copy files that describes the project and its dependencies
COPY package.json package-lock.json /

# Install dependencies
RUN NODE_ENV=production npm ci

# Copy the source code
COPY server.js .

# Setup environment
ENV PORT=80

# Define the script used for running the application
CMD [ "npm", "start" ]
```

Figure 6. HTTP service Dockerfile

3.3.3 Queue consumer function

To process any messages published on the event bus by the HTTP service, a function had to be implemented. However, before that, a list of requirements for the function was collected. This list is shown in Table 5.

Table 5. Requirements for message consumer

Unique identifier	**Requirement**
2.1	The function should be able to process multiple messages with one invocation
2.2	The function should create a new file on S3 containing the UUID
2.3	The function should change stage for the given document in the database to 'processed'
2.4	The function should set the property 'processedAt' to the current time

Just like the HTTP service, a new npm project was created in a new directory. The *aws-sdk* was installed, as it would be needed to communicate with S3 and DynamoDB. After that, a file called function.js was created to contain the code needed to run the function. The resulting code in function.js is shown in Figure 7.

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dbClient = new AWS.DynamoDB.DocumentClient()

exports.handler = async (event) => {
  for (const record of event.Records) {
    const id = JSON.parse(record.body).Message
    await s3.putObject({
      Body: id,
      ContentType: 'text/html',
      Bucket: process.env.S3_BUCKET,
      Key: `${id}.txt`
    }).promise()

    await dbClient.update({
      TableName: process.env.TABLE_NAME,
      Key: {
        id
      },
      UpdateExpression: 'set processedAt = :p, stage = :s',
      ExpressionAttributeValues: {
        ':p': Date.now(),
        ':s': 'processed'
      },
      ReturnValues: 'NONE'
    }).promise()
  }
  return {}
}
```

Figure 7. Consumer function code

First, the AWS SDK was included and initialized to allow for communication with both S3 and DynamoDB. Then the actual function handler was implemented. On invocation, the function would receive an event object containing all messages in the Records property. These records were iterated over and processed one by one. In each iteration, the id was parsed, a simple txt-file containing the id was put on S3, and the corresponding document in the database updated according to requirements 2.3 and 2.4. To communicate with S3 and DynamoDB, bucket name, respectively table name, had to be given. These could potentially change on every deployment, so just as with the HTTP service, environment variables were utilized.

3.3.4 Background job function

To allow for running recurring tasks, such as cleanup of old resources to be GDPR compliant, another function would have to be implemented. Just as for the other applications, a list of requirements was first collected. This list is shown in Table 6.

Table 6. Requirements for job function

Unique identifier	Requirement
3.1	The function should select all documents that had been processed and were older than one hour
3.2	The function should delete files on S3 associated with old documents
3.3	The function should change stage for the old documents in the database to 'deleted'
3.4	The function should set the property 'deletedAt' to the current time

The function handler was created in the same way as the consumer function: a new npm project was created in a new directory. The *aws-sdk* was installed to allow for communication with S3 and DynamoDB. All code were put in a file called function.js, which is shown in Figure 8.

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dbClient = new AWS.DynamoDB.DocumentClient()

exports.handler = async () => {
  const data = await dbClient.query({
    TableName: process.env.TABLE_NAME,
    IndexName: process.env.GSI_INDEX,
    ProjectionExpression: 'id',
    KeyConditionExpression: 'stage = :s and createdAt <= :d',
    ExpressionAttributeValues: {
      ':s': 'processed',
      ':d': Date.now() - 60 * 60 * 1000
    }
  }).promise()

  for (const { id } of data.Items) {
    await s3.deleteObject({
      Bucket: process.env.S3_BUCKET,
      Key: `${id}.txt`
    }).promise()

    await dbClient.update({
      TableName: process.env.TABLE_NAME,
      Key: {
        id
      },
      UpdateExpression: 'set deletedAt = :d, stage = :s',
      ExpressionAttributeValues: {
        ':d': Date.now(),
        ':s': 'deleted'
      },
      ReturnValues: 'NONE'
    }).promise()
  }
  return {}
}
```

Figure 8. Job function code

At the beginning of the code, the AWS SDK was included and initialized as usual. The handler function was defined, but in this case, the event argument was omitted. This was because the actual content of the event did not matter for this function; it just had to be invoked by any event.

Inside the handler, the database was first queried for any documents in the processed stage that were older than one hour, indicated in milliseconds. Since only the id-property was needed, all other properties were excluded to lower load on both the database and the function's memory. Also, since these query conditions were not based on the primary key, 'id', a specific index had to be used. This index was a so-called global secondary index (GSI) and will be further explained in 3.3.5.4.

For each found document id, the associated file on S3 was deleted and the document updated to fulfill requirements 3.2 through 3.4. To invoke the functions required to delete on S3 and update on DynamoDB, dynamic variables had to be supplied, which were taken care of by yet again utilizing environment variables. It was also decided to put the name of the GSI in an environment variable to allow for infrastructure naming changes without modifying the code.

3.3.5 Infrastructure implementation

With all application-level code written, it was time to implement the infrastructure. This section will describe that process, the decisions that were made, and why. Just as with the HTTP service and functions, code will be shown. However, instead of showing all code in one figure, it will be divided into multiple sections for easier comprehension.

3.3.5.1 Initialization

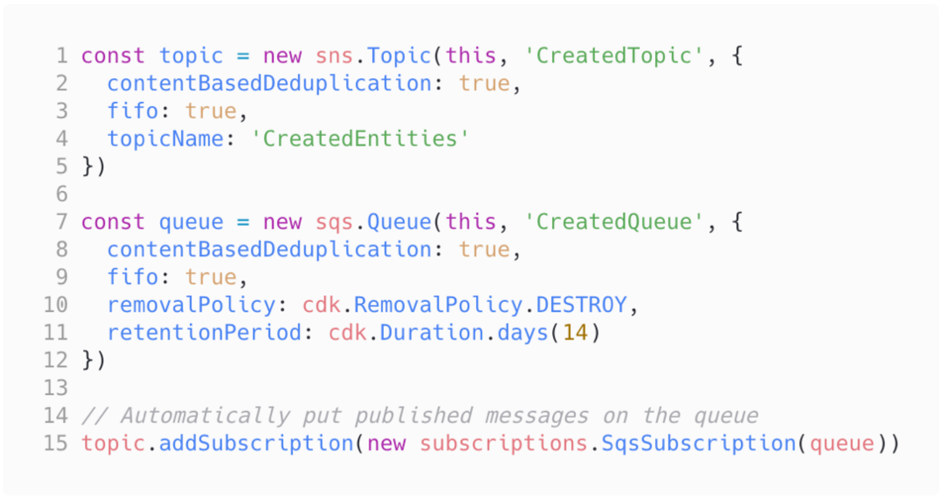
To start, an empty directory was created and initialized using the CDK CLI. JavaScript was the chosen language for the infrastructure to make for a seamless transition between application code and infrastructure code. The HTTP service, queue consumer, and task function were moved into the subdirectory *services* to allow for automatic deployment of application-level changes.

3.3.5.2 Creating Event bus and Queue

The event bus used to communicate between the HTTP service and the consumer function consists of one SNS topic and one SQS. To create and connect these, the following dependencies had to be installed:

- @aws-cdk/aws-sns
- @aws-cdk/aws-sqs
- @aws-cdk/aws-sns-subscriptions

These modules were then used to create and connect the topic and queue, as shown in Figure 9.



```javascript
const topic = new sns.Topic(this, 'CreatedTopic', {
  contentBasedDeduplication: true,
  fifo: true,
  topicName: 'CreatedEntities'
})

const queue = new sqs.Queue(this, 'CreatedQueue', {
  contentBasedDeduplication: true,
  fifo: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  retentionPeriod: cdk.Duration.days(14)
})

// Automatically put published messages on the queue
topic.addSubscription(new subscriptions.SqsSubscription(queue))
```

Figure 9. SNS and SQS setup

As shown in the figure, both the queue and topic were configured with the flags *contentBasedDeduplication* and *fifo* set to true. Although the concept of FIFO is described in 2.4.8 and 2.4.9, content-based deduplication is not. Enabling this ensures that even if the same message is posted to the topic more than once within 5 minutes, only the first message is kept [48].

Since SQS persists data, two more properties were configured. *removalPolicy* is a property that can be configured on many AWS services and allows the developer to decide what should happen with a specific service when the CDK stack is destroyed. For the sake of making developing easier, this was set to *destroy*, which deletes the queue along with all its content. In a real production environment, it would probably be more desirable to use *retain*, which will keep the queue even if the stack is destroyed. *rentationPeriod* was set to the maximum, 14 days, to keep unprocessed items in the queue for as long as possible.

To automatically put messages published to the topic on the queue, *aws-sns-subscriptions* were utilized.

3.3.5.3 Creating an S3 bucket

Since object storage was needed for the consumer function, a simple storage service (S3) bucket was created, as shown in Figure 10. To create the bucket, a new dependency had to be installed: @aws-cdk/aws-s3.

```
const bucket = new s3.Bucket(this, 'EntityBucket', {
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true,
  publicReadAccess: false
})
```

Figure 10. S3 setup

Just like for the SQS, S3 was configured with *removalPolicy* set to *destroy*. However, with S3, this only deletes the bucket if it is empty. So, to avoid manual deletion of objects, the property *autoDeleteObjects* was set to true. As mentioned with SQS, this is not recommended in a production environment, as critical data might be lost unexpectedly. Besides that, *publicReadAccess* was disabled to ensure that any files stored in the bucket only were accessible by services with explicitly granted permissions.

3.3.5.4 Creating DynamoDB database

As DynamoDB was the chosen real-time database, a DynamoDB table was created. For this, the dependency @aws-cdk/aws-dynamodb was installed. The resulting code is shown in Figure 11.

```
const table = new dynamodb.Table(this, 'Entities', {
  partitionKey: { name: 'id', type: dynamodb.AttributeType.STRING },
  removalPolicy: cdk.RemovalPolicy.DESTROY
})

const indexName = 'stage-created-index'
table.addGlobalSecondaryIndex({
  indexName,
  partitionKey: { name: 'stage', type: dynamodb.AttributeType.STRING },
  sortKey: { name: 'createdAt', type: dynamodb.AttributeType.NUMBER }
})
```

Figure 11. DynamoDB setup

The table was configured with *removalPolicy* set to *destroy*, as previously motivated for SQS and S3. Also, the partition key (and primary key) *id* was specified. This was the attribute that would contain the UUID generated by the HTTP service. This made the choice to use a UUID even more apparent, as the primary key needs to be unique and is used internally by DynamoDB to resolve what physical machine the document is stored on [49].

But since the task function needed to efficiently find documents based on their stage and when they were created, just using the primary key would not suffice. Hence, a global secondary index (GSI) was introduced [50]. The attribute *stage* was chosen for the partition key as it would only be queried by exact match. To allow for range queries such as less than and greater than, an optional sort key was also added to the GSI. The attribute *createdAt* was chosen for the sort key as the task function queries based on the creation timestamp.

3.3.5.5 Creating Lambda function for queue consumer

To process the queue, a Lambda function had to be provisioned. A VPC was also created at this point since a Lambda function can be placed within one and thus allowing it to be private. The final code used is shown in Figure 12. To create the VPC, the Lambda function and register it to listen to the queue, the following dependencies had to be installed:

- @aws-cdk/aws-ec2
- @aws-cdk/aws-lambda
- @aws-cdk/aws-lambda-event-sources

```
const vpc = new ec2.Vpc(this, 'Vpc')
const queueConsumer = new lambda.Function(this, 'QueueConsumer', {
  runtime: lambda.Runtime.NODEJS_14_X,
  handler: 'function.handler',
  code: lambda.Code.fromAsset('./services/queue-consumer'),
  environment: {
    S3_BUCKET: bucket.bucketName,
    TABLE_NAME: table.tableName
  },
  vpc,
  memorySize: 128
})

// Register queue consumer to process queue
queueConsumer.addEventSource(new SqsEventSource(queue, {
  batchSize: 10
}))
```

Figure 12. Queue Consumer Lambda setup

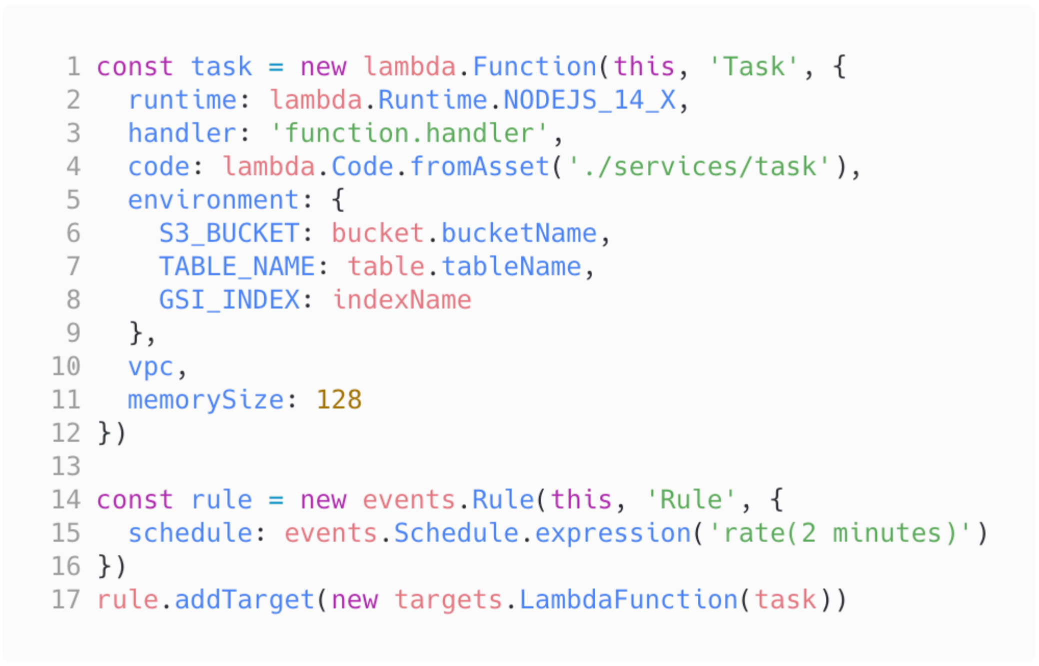
First, the Lambda was configured to use the Node 14.X. This specific version was chosen as it was the official long-term support (LTS) version at that time. The *handler* property was set to *function.handler*. The first part denotes what file to look for, *function.js*, and the second part at what exported attribute in that file the function will be available. The *code* attribute was also specified. It points to the directory in which the handler exists. Using the *fromAsset* function, CDK will automatically take the contents of that folder, archive it to a .zip-file, and upload it to Lambda. The created VPC was also linked, and the minimum memory size was chosen to keep costs as low as possible.

As mentioned in 3.3.3, the function was designed to depend on environment variables to configure the AWS services. The needed variables for S3 and DynamoDB were defined and referenced from the previously created bucket and table instances. It was, therefore, crucial that the Lambda was created after both the bucket and the table so that the references would be initialized during deployment.

The last step was to register the newly created Lambda to listen for events from the queue. This was done with the help of *aws-lambda-event-sources*. Since the function was built to support multiple events at once, the event source was configured to deliver the maximum allowed batch size.

3.3.5.6 Creating Lambda function for recurring task

Since the configuration of this function was much similar to the previously created Lambda function, only the differences will be discussed in this section. The resulting code is shown in Figure 13.



```
const task = new lambda.Function(this, 'Task', {
  runtime: lambda.Runtime.NODEJS_14_X,
  handler: 'function.handler',
  code: lambda.Code.fromAsset('./services/task'),
  environment: {
    S3_BUCKET: bucket.bucketName,
    TABLE_NAME: table.tableName,
    GSI_INDEX: indexName
  },
  vpc,
  memorySize: 128
})

const rule = new events.Rule(this, 'Rule', {
  schedule: events.Schedule.expression('rate(2 minutes)')
})
rule.addTarget(new targets.LambdaFunction(task))
```

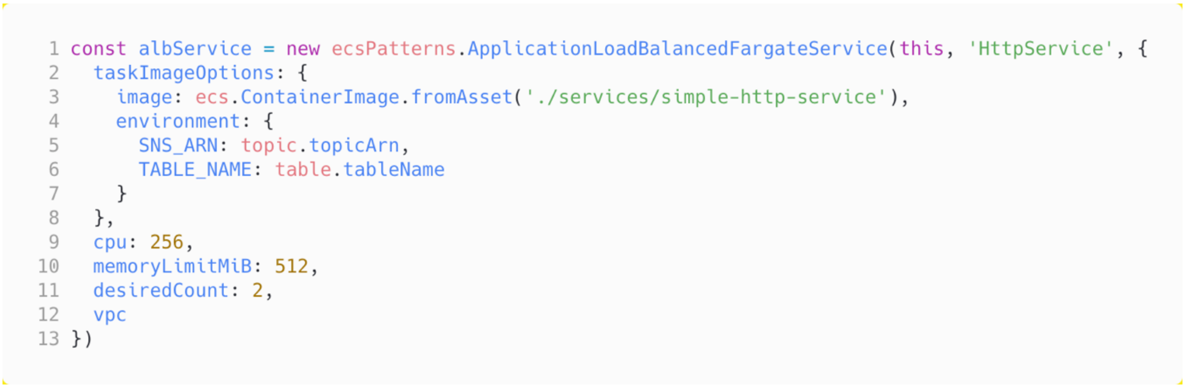
Figure 13. Task Lambda setup

Configuration of the Lambda instance was precisely the same as for the queue consumer, with the difference that another directory containing the function code was selected and that a third environment variable, *GSI_INDEX*, was injected.

The difference compared to the queue consumer was in how the function got triggered in the first place. For this Lambda, the goal was to trigger it based on a schedule. To accomplish this, two new dependencies were installed: @aws-cdk/aws-events and @aws-cdk/aws-events-targets. Using these, a recurring CloudWatch event was created and registered through AWS EventBridge [51] as the trigger for the Lambda function.

3.3.5.7 Creating a load-balanced Fargate cluster

In the previous sections related to writing infrastructure as code, one was created for each AWS service. For this part of the infrastructure, a higher-level construct found in *@aws-cdk/aws-ecs-patterns* [52] was used. This was chosen to reap the benefits of best practice design without writing any unnecessary code. The resulting code is shown in Figure 14.



```
const albService = new ecsPatterns.ApplicationLoadBalancedFargateService(this, 'HttpService', {
  taskImageOptions: {
    image: ecs.ContainerImage.fromAsset('./services/simple-http-service'),
    environment: {
      SNS_ARN: topic.topicArn,
      TABLE_NAME: table.tableName
    }
  },
  cpu: 256,
  memoryLimitMiB: 512,
  desiredCount: 2,
  vpc
})
```

Figure 14. Fargate service setup

Using the pattern *ApplicationLoadBalancedFargateService*, an Application Load Balancer was automatically created and linked with the Fargate service. However, some configurations still had to be made. The *image* property was specified just as the *code* property was specified for the Lambda functions, with the difference that the *ContainerImage* class automatically builds an image in the selected directory and pushes it to ECR on deployment. This image is then used by Fargate when starting the resulting container(s).

On top of that, the service was connected to the previously created VPC and configured to run two instances with the smallest possible CPU and memory options.

To cope with variable load, automatic scaling was configured using the code shown in Figure 15. The service was configured such that CPU usage would determine whether to scale out or scale in, but any metric observed by CloudWatch could be used to trigger automatic scaling. In the background, CloudFormation creates two CloudWatch alarms: one that triggers scale-out when CPU usage is 70 percent or over for three consecutive periods, and one that triggers scale-in when CPU usage is below 70 percent for 15 consecutive periods. For the quickest possible scaling, the period length was set to 60 seconds for both scale-out and scale-in.

```
const scalableTarget = albService.service.autoScaleTaskCount({
  minCapacity: 2,
  maxCapacity: 10,
})

scalableTarget.scaleOnCpuUtilization('CpuScaling', {
  targetUtilizationPercent: 70,
  scaleInCooldown: cdk.Duration.seconds(60),
  scaleOutCooldown: cdk.Duration.seconds(60),
})
```

Figure 15. Auto-scaling setup

3.3.5.8 Handling permissions

Arguably one of the most critical parts of infrastructure is its security. To follow the principle of least privilege [53], services were only granted the explicit permissions needed to complete their task. The resulting code is shown in Figure 16.

```
// HTTP service permissions
topic.grantPublish(albService.taskDefinition.taskRole)
table.grantWriteData(albService.taskDefinition.taskRole)

// Consumer function permissions
bucket.grantPut(queueConsumer.role)
table.grantWriteData(queueConsumer.role)

// Task function permissions
bucket.grantDelete(task.role)
table.grantReadWriteData(task.role)
```

Figure 16. Permission setup

3.4 Phase 4: Evaluation

In this phase, the infrastructure and the applications that should run on it were provisioned on AWS. After that, it was evaluated based on measurements and observations.

3.4.1 Deploying and testing

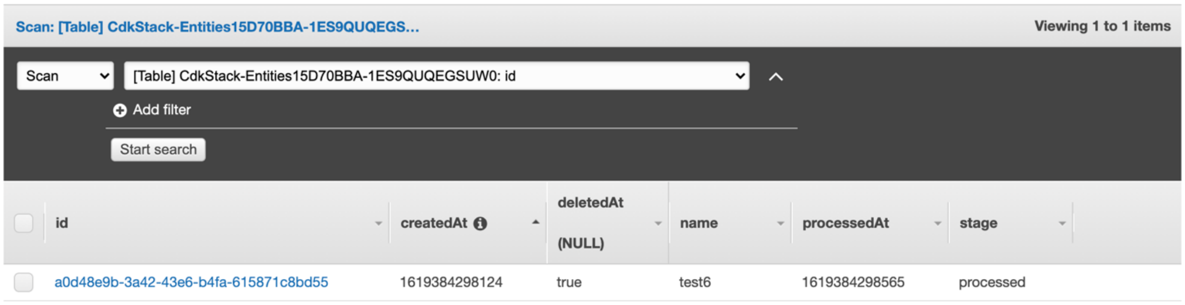
After writing all application and infrastructure code, it was time to deploy it to AWS. By running *cdk deploy*, the infrastructure and its applications were automatically built and deployed. The time it took to deploy everything from scratch was measured five times using the built-in program *time* [54]. The measurements are presented in Table 7.

Table 7. Deploy duration from scratch

Try	**Duration (minutes: seconds)**
1	6:30
2	6:31
3	6:29
4	6:28
5	6:32
Average	**6:30**

With everything running on AWS, it was time to test the application. The HTTP service was called on the URL produced by the deployment, using Google Chrome. Calling the endpoint multiple times revealed that the service was indeed running on two hosts: *ip-10-0-179-51.eu-west-1.compute.internal* and *ip-10-0-252-252.eu-west-1.compute.internal*.

To determine if the application was working, DynamoDB was inspected first using the AWS console. Everything looked as intended – one row had been created, and all fields were populated with expected values, see Figure 17. The difference between the fields *createdAt* and *processedAt* shows how long it took for the event to get processed by the consumer function. In this case, it was 441 milliseconds. Since the consumer function also produces and uploads a .txt file on S3; that too was verified using the AWS console.


Scan: [Table] CdkStack-Entities15D70BBA-1ES9QUQEGS... Viewing 1 to 1 items

Scan | [Table] CdkStack-Entities15D70BBA-1ES9QUQEGSUW0: id

Add filter

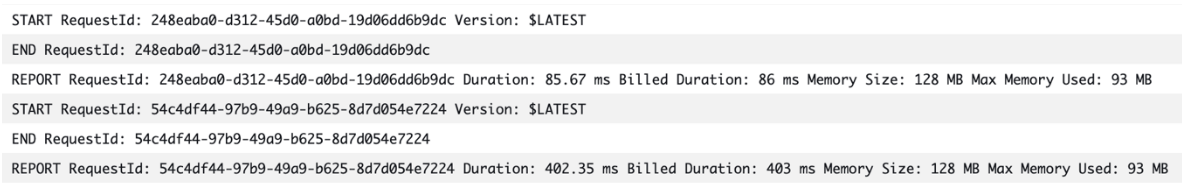
Start search

id	createdAt	deletedAt (NULL)	name	processedAt	stage
a0d48e9b-3a42-43e6-b4fa-615871c8bd55	1619384298124	true	test6	1619384298565	processed

Figure 17. DynamoDB console after creation

After waiting one hour, the background job could also be verified. Refreshing the DynamoDB interface showed that the stage was updated to *deleted* and *deletedAt* set to a valid timestamp. On top of that, the .txt file associated with the database document was automatically removed.

CloudWatch was also examined to ensure that logs were captured for both the service running on Fargate and the functions running on Lambda. Figure 18 shows some logs automatically produced by two invocations of the task function. The last run was much slower since the document had just gotten older than one hour, and actions had to be taken.


```
START RequestId: 248eaba0-d312-45d0-a0bd-19d06dd6b9dc Version: $LATEST
END RequestId: 248eaba0-d312-45d0-a0bd-19d06dd6b9dc
REPORT RequestId: 248eaba0-d312-45d0-a0bd-19d06dd6b9dc Duration: 85.67 ms Billed Duration: 86 ms Memory Size: 128 MB Max Memory Used: 93 MB
START RequestId: 54c4df44-97b9-49a9-b625-8d7d054e7224 Version: $LATEST
END RequestId: 54c4df44-97b9-49a9-b625-8d7d054e7224
REPORT RequestId: 54c4df44-97b9-49a9-b625-8d7d054e7224 Duration: 402.35 ms Billed Duration: 403 ms Memory Size: 128 MB Max Memory Used: 93 MB
```
Figure 18. CloudWatch logs for background task

3.4.2 Measuring autoscaling
Since Zmarta currently has no way to set up automatic scaling, this topic was thoroughly measured and evaluated, which is described in this section.

To benchmark the application with and without automatic scaling, a script was built in Node.js. The library *AutoCannon* [55] was utilized to simplify making concurrent requests and capturing certain data. The complete code for the script can be found in Appendix 8.1.

Automatic scaling was disabled for the first benchmark and enabled for the second. Both tests ran for exactly 15 minutes with ten concurrent requests. Using the .csv file created by the script, the throughput could be visualized, see Figure 19.

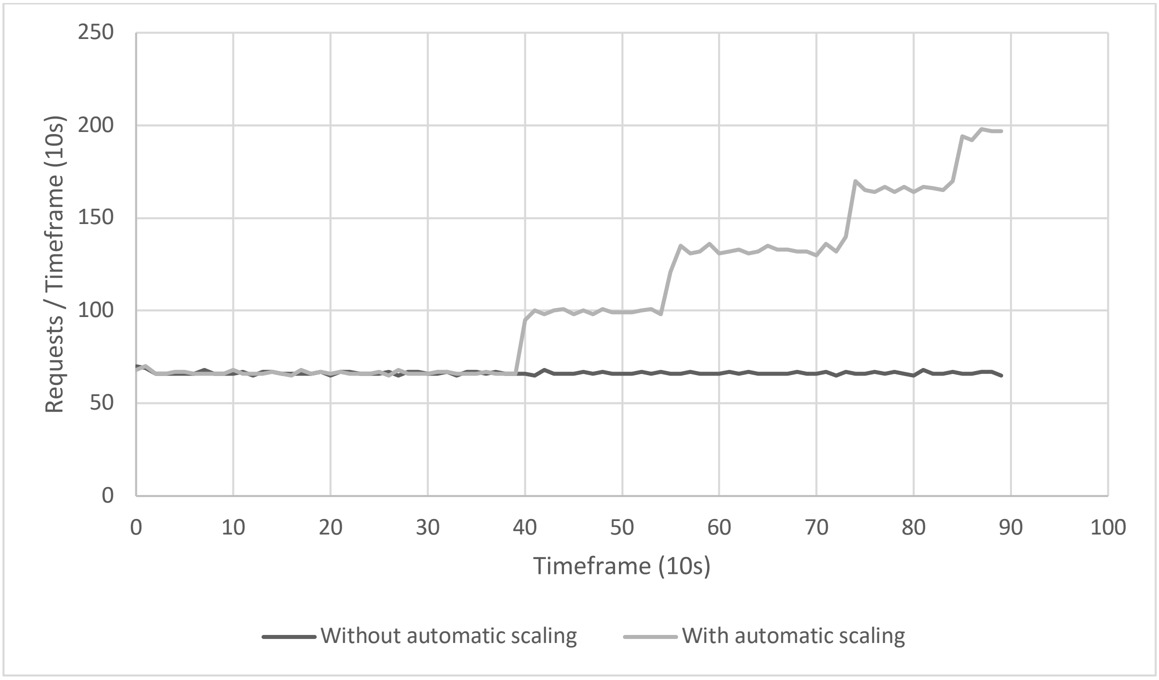


Figure 19. Throughput comparison

During the first six minutes, there was no difference to enabling automatic scaling. However, this is expected. Since the average CPU usage is used as the scaling metric, it took some time to exceed 70 percent. After that, instances were automatically added one by one, clearly indicated by the sudden increases in throughput.

This increase in available resources resulted in considerably more processed requests, with a lower average latency, shown in Table 8. Keep in mind that these numbers are the averages over the whole duration, meaning that the actual latency was much lower with the service scaling out.

Table 8. Other benchmark numbers

Metric	With scaling	Without scaling
Average latency	947.61 milliseconds	1503.72 milliseconds
Total requests	9488	5979

3.4.3 Deploying changes

In the real world, most deploys will be changes to an existing application. To test that, changes were made to the response text of the HTTP service and deployed using *cdk deploy*. The *time* application was also used this time to measure the duration. This was repeated five times, and the measurements are shown in Table 9. The HTTP service was called after each deployment to verify that the changes were live.

Table 9. Deploy duration for small change

Try	**Duration (minutes: seconds)**
1	2:58
2	2:59
3	3:01
4	2:57
5	2:57
Average	**2:58**

The average time was quite fast, considering that *ApplicationLoadBalancedFargateService* automatically uses a rolling deployment strategy [56]. This means that old containers are slowly and safely drained and replaced by the new version, one by one.

Using the benchmarking script described in 3.4.2, measurements were collected during a deployment. The script was configured to run ten concurrent requests for 6 minutes as the average deployment took 3 minutes. No requests were dropped during the process, and the throughput remained high. Figure 20 clearly shows a spike at the 3-minute mark where all four instances are running. Shortly after, the old instances were drained, and the throughput went back to normal.

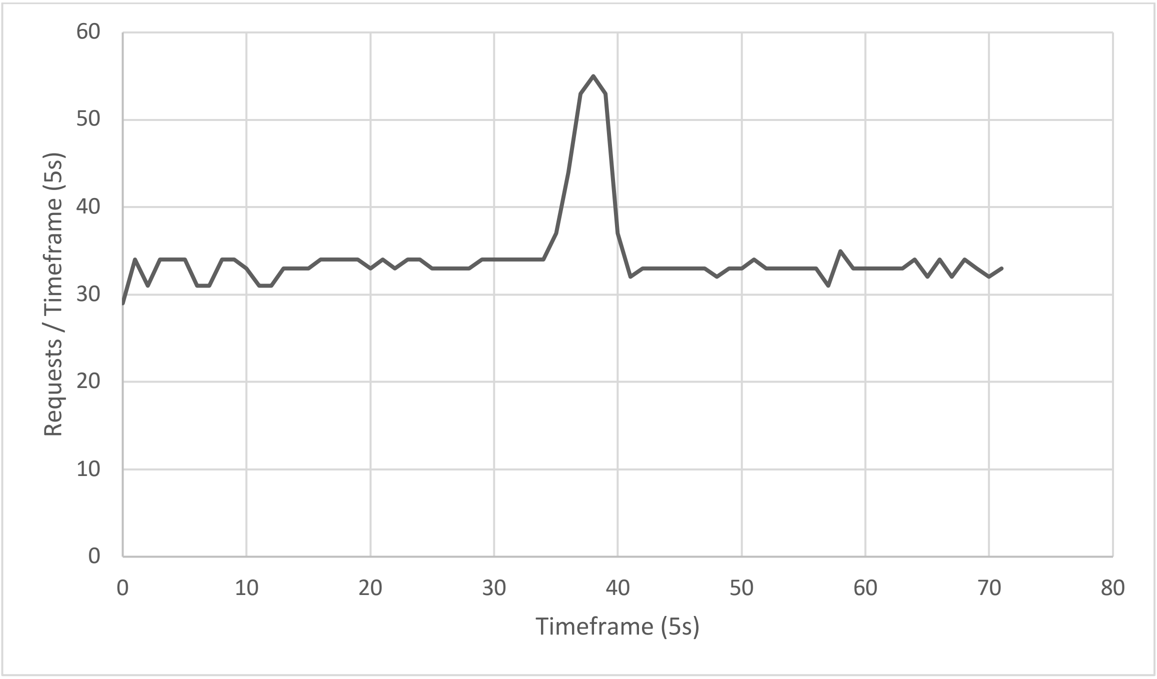


Figure 20. Throughput during deploy

3.4.4 Teardown

With all testing and measurements being completed, the infrastructure was taken down not to waste resources. Using *cdk destroy*, everything was deleted, including data persisted in DynamoDB and S3. This command was repeated and measured five times using, yet again, the *time* application. The results are shown in Table 10.

Table 10. Destroy duration

Try	Duration (minutes: seconds)
1	3:05
2	3:01
3	2:59
4	3:03
5	3:02
Average	**3:02**

3.5 Sources

This section describes the different groups of sources used and why they can be trusted. All related sources will be referenced at the end of each subsection.

3.5.1 Amazon

Amazon is the most used source in this thesis, both in documentation, articles, interviews, and presentations. Amazon is a well-known and established cloud service vendor that benefits from educating potential customers, and since only documentation and general concepts were cited, that can be trusted. These sources are: [1] [8] [16] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [35] [37] [44] [45] [48] [49] [50] [51] [52] [56].

3.5.2 Microsoft

Microsoft Azure's official website was also used to collect information on general topics in the cloud. Like Amazon, Microsoft is well-established and can therefore be trusted when it comes to general educational topics. Following sources from Microsoft were used: [2] [3] [4] [5].

3.5.3 IBM

IBM is also a well-established tech organization, which makes any non-marketed information trustworthy. Used sources: [6] [7] [15].

3.5.4 Official software documentation

Many different official software sites were used to gather information about what the software does and how. However, mainly their documentation was used to accomplish all steps. Since they were only used to describe that specific software and not to compare it with its potential competitors, there is no conflict of interest, and the material can be trusted. These sources are: [9] [10] [11] [12] [14] [17] [34] [41] [42] [43] [46] [47] [54] [55].

3.5.5 Peer-reviewed publications

Some information was fetched from peer-reviewed articles and conference papers published by trusted bodies like IEEE. Since experts on the topic must review all material before being posted, these sources are very trustworthy. Following sources are from reviewed publications: [13] [36] [38] [39] [53].

3.5.6 Course literature

Books used as course literature at LTH were used to collect information on development methodology and elicitation techniques. The fact that they are used as course literature by a respected educational institution such as LTH means that they can be trusted. These sources are: [33] [40].

4 Results

This section presents and explains the results obtained in the Methodology and Analysis chapter. As all written code is a big part of this thesis work, it can be found in a raw format in Appendix 8.2, 8.3, 8.4 and 8.5.

4.1 Infrastructure provisioning

Based on the work in Phase 1, the cloud infrastructure was designed in Phase 2 and then implemented in Phase 3. Since all infrastructure was written as JavaScript code using Cloud Development Kit, discussed in section 3.3.5, changes could easily be tracked using a file versioning tool like Git [17].

Also, since CDK was used to write the infrastructure, both application- and infrastructure-level changes could be automated. By running the deploy command, listed in Table 3, CDK would automatically detect changes in either infrastructure or associated applications and then push the changes to AWS.

4.2 AWS Services

In Phase 1, an interview was conducted with Jonas, Zmarta's Architect. He recommended using AWS Secrets Manager to store and deliver sensitive data such as database credentials and API keys to the different applications. Initially, Secrets Manager was included in the infrastructure design. However, during implementation using the CDK, it was found out that these secrets could be securely injected during deployment using environment variables. Although Secrets Manager could still be used, there was no need when CDK could automate it out of the box.

To allow for highly scalable applications, an event-driven architecture was implemented. AWS's Simple Notification Service and Simple Queue Service were used to delivering events to consumers to facilitate this type of architecture. In this thesis, only one consumer was designed to illustrate the point, and it was implemented as a Lambda function. A Lambda function was also used to create a recurring background job.

Since HTTP was the decided entry point for requests, a Fargate cluster running an HTTP application written in Node.js was created. Using Fargate, it was possible to configure automatic scaling without carrying about the underlying machines. To distribute the load between each application instance running in Farage, an Application Load Balancer was used. The ALB was also used to automatically transition traffic to new containers during deployment, allowing for a rolling deployment.

4.3 Measurements

To evaluate certain aspects of the cloud infrastructure, measurements were performed. However, no measurements were performed on Zmarta's current infrastructure as it could have affected their systems in ways not worth risking. To make up for this, Zmarta's head of IT operations Erik mentioned a few pitfalls of their current infrastructure that could be evaluated in AWS.

4.3.1 Deployment
One core part of the daily development lifecycle at Zmarta is continuous delivery. For this to work, quick and reliable deployments are a must. Measurements collected in Phase 4 show that deployments of application-level changes consistently took around 3 minutes, as shown in Table 9. Also, throughput never decreased, and no requests failed during any deployment, which is visualized in Figure 20. This is a big improvement compared to Zmarta's current infrastructure as the same process takes 5 to 15 minutes depending on how many developers are deploying at that very moment. It also solves their problem which sometimes causes a handful of requests to fail during a deployment.

The time it took to deploy the whole infrastructure from scratch was also measured. The average measured time was 6 minutes and 30 seconds, with only a few seconds of deviation, as shown in Table 7. This is much faster than the 1 to 2 hours Erik Holmqvist, head of IT operations at Zmarta, estimated it would take doing it manually. Although this is not a common thing to do, it illustrates how easy it would be for Zmarta to expand this infrastructure if they would start to offer their products in a new country. On top of that, since there is no human factor involved in that process, each expansion would be 100 percent consistent with all other deployments.

While not very relevant for production environments, the time it took to tear down the infrastructure was also measured. This is still important, as a quick and easy teardown allows for experimentation in a testing environment without the overhead of manually destroying each service not to incur unnecessary costs. Teardown of the infrastructure took consistently around 3 minutes, as shown in Table 10.

4.3.2 Automatic scaling
Since Zmarta had no support for automatic scaling with their current infrastructure, this became an important topic to explore. By writing a few lines of infrastructure code, shown in Figure 15, it was possible to enable this in AWS. To measure the impact of said autoscaling, a program was written to perform multiple concurrent requests while measuring the outcome and duration of each. Figure 19 clearly shows the throughput being increased discretely with each new instance being started. Although not shown by the measurements, it is safe to assume that, if desired, a practically endless scaling would be possible given AWS's vast resources.

On top of throughput, average latency was also measured, shown in Table 8. Using automatic scaling resulted in the average latency decreasing by more than half a second. This is a big deal for someone like Zmarta that prioritizes customer experience.

5 Conclusions

This section summarizes the most important results and explicitly answers all questions stated in section 1.4. The conclusions will serve as an example for Zmarta on how their infrastructure could be moved to a cloud-native solution. On top of that, ethical aspects and future work are discussed.

The purpose was fulfilled by investigating Zmarta's needs and then implementing a comparable infrastructure on AWS with a Could-Native approach using Infrastructure as Code. The resulting setup managed to perform better than their current infrastructure in all selected metrics and provides automatic scaling, which is something they currently do not support.

5.1 Fulfilling the purpose

5.1.1 Research question 1

The question "How can infrastructure be provisioned in the cloud in an efficient and traceable manner?" was answered by the infrastructure code produced in Phase 3. By writing the infrastructure in a declarative manner using the same language used for developing the applications, JavaScript, it was possible to define the whole infrastructure in 135 lines of code, including spaces and comments for readability.

The process can be considered efficient as it eliminates any manual configuration in AWS's console and allows for identical deployments in different geographical regions by changing only one line of code. Moreover, the fact that no manual configuration is needed means that the code is the single source of how the configuration is done and can be tracked by a versioning tool such as Git.

5.1.2 Research question 2

Using AWS's Cloud Development Kit, it was possible to deploy multiple application-level changes with one command: *cdk deploy*. The fact that CDK was utilized and can be used to deploy using only one command answers the question of "How can the release of application-level changes be automated?".

5.1.3 Research question 3

During the interview with Zmarta's head of IT operations, Erik, in Phase 1, it was discovered that the initial question "What is the difference between different levels of managed services in terms of cost and features?" was not relevant to explore. Although fully managed services usually are slightly more expensive, Erik explained that they almost always save the company money in the end by not requiring as much developer time. Since this question cannot be answered scientifically in a reasonable manner, it was decided that fully managed services would be used where possible, focusing on making it as easy as possible to work with the infrastructure.

5.1.4 Research question 4

With the use of CDK, ten lines of code were written to enable automatic scaling based on CPU usage. These ten lines, along with the measurements performed to evaluate the automatic scaling, answers the question of "How can autoscaling be configured and utilized to adapt for a variable load?". It was clearly shown in the results that applications running on AWS Fargate could automatically scale, practically without any boundaries.

5.1.5 Research question 5

To answer the question "Which AWS services are appropriate for fulfilling the company's needs?" AWS and its services were studied to find ones appropriate for Zmarta's needs.

This resulted in Fargate being chosen for running the HTTP application with an Application Load Balancer in front. Simple Notification Service and Simple Queue Service were used to deliver events to consumers to facilitate an event-driven architecture. Lambda functions were used to consume asynchronous events published on SNS and running recurring tasks based on a schedule. Any data was stored in DynamoDB and Simple Storage Service.

To write and manage the infrastructure, a few other services were utilized. Cloud Development Kit was used to write Infrastructure as Code, automatically taking care of deploying changes in both infrastructure and applications. In the background, CDK used Elastic Container Registry to store containers that would run on Farage. To monitor this, logs were pushed and visualized in CloudWatch.

5.1.6 Research question 6 and 7

By adopting the cloud-native solution proposed by this thesis, Zmarta would get a highly extensible infrastructure that can automatically scale to fit current load conditions. Everything was written in JavaScript, meaning that all Zmarta developers could read and write infrastructure in a language that they are comfortable with. It also means any changes to the infrastructure could be tracked using Git.

To adopt this solution, Zmarta would need to migrate all services from their current infrastructure, which would most likely take a lot of time and risk breaking things. Zmarta would also lose the lower-level control they currently have by running their applications on Elastic Compute instances, which might be a problem if they want to solve problems they might encounter in the future using a specific piece of software.

5.1.7 Research question 8

What comparisons and measurements should be performed to evaluate the architectures depends on the company's values running these architectures. For Zmarta, their quality of service must be high. Therefore, the topic of automatic scaling was measured in terms of throughput and request latency. It is also crucial for the company that software can consistently be delivered fast and reliably to production. The time it took to perform deployments and how requests were handled during deployments were measured.

5.2 Ethical aspects

Since Zmarta provides services for making comparisons in the financial sector, their systems handle sensitive and confidential data such as social security numbers and credit reports. This is especially important with the restrictions put in place in the EU with GDPR.

By using the proposed solution, all data is stored in services managed by AWS. With all its resources, AWS can be expected to detect and fix potential security vulnerabilities faster than could ever be possible by Zmarta employees managing an in-house solution.

The proposed solution also ensures that no data is communicated over a public network if it does not need to be. Using AWS's Virtual Private Cloud, all service-to-service communication was configured to be routed through a private subnet, only putting the load balancer in the publicly accessible subnet.

Another critical factor when it comes to data security is the principle of least privilege. By only granting services the minimum permission(s) needed to fulfill their task, potential data exposure is minimized. In the proposed solution, each service was explicitly granted the permissions they needed. Furthermore, since the permissions were granted declaratively using code, they could be put through a code-review process, making it more improbable that wrong permissions get granted by mistake.

5.3 Future work

This thesis work only scratches the surface of what could be accomplished using a cloud-native solution. One thing that would be interesting to investigate is the differences between the big cloud providers Google Cloud, Azure, and AWS. This could be useful to optimize cost and performance. Knowing how to work with different providers could aid in building a multi-cloud solution, which could help minimize vendor lock-in.

It would also be interesting to delve deeper into the topic of deployment on AWS. In this thesis, a rolling deployment was used. However, more advanced techniques such as blue-green deployments and canary releases could be studied to help minimize the risks associated with deployments.

6 Terminology

Cloud-Native – A way to design systems taking advantage of cloud services.

Containerization – The process of packaging applications and their required environment in an image that can run consistently on any machine.

Event-Driven Architecture – A architecture paradigm that allows for decoupling of applications. It consists of publishers, a messaging bus, and consumers at its core.

IaaS – Stands for *Infrastructure as a Service*.

IaC – Stands for *Infrastructure as Code* and is the process of writing infrastructure with code instead of traditional configuration using a CLI or graphical interface.

PaaS – Stands for *Platform as a Service*.

Serverless – A cloud computation model used to categorize services that scales on-demand.

SaaS – Stands for *Software as a Service*.

VPS – Stands for *Virtual Private Server.*

7 References

[1] "What is AWS," Amazon, [Online]. Available: https://aws.amazon.com/what-is-aws/. [Accessed 26 March 2021].

[2] "What is IaaS?," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-iaas/. [Accessed 27 March 2021].

[3] "What is PaaS?," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-paas/. [Accessed 27 March 2021].

[4] "Serverless computing," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/overview/serverless-computing/. [Accessed 27 March 2021].

[5] "What is SaaS?," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-saas/. [Accessed 27 March 2021].

[6] "Containerization," IBM Cloud Education, 15 May 2019. [Online]. Available: https://www.ibm.com/cloud/learn/containerization. [Accessed 28 March 2021].

[7] "Containerization," IBM Cloud, 18 March 2019. [Online]. Available: https://www.youtube.com/watch?v=0qotVMX-J5s. [Accessed 28 March 2021].

[8] "What is Docker," Amazon, [Online]. Available: https://aws.amazon.com/docker/. [Accessed 28 March 2021].

[9] "Dockerfile reference," Docker, [Online]. Available: https://docs.docker.com/engine/reference/builder/. [Accessed 28 March 2021].

[10] "Docker overview," Docker, [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed 28 March 2021].

[11] "Docker Registry," Docker, [Online]. Available: https://docs.docker.com/registry/. [Accessed 28 March 2021].

[12] "Docker Hub Quickstart," Docker, [Online]. Available: https://docs.docker.com/docker-hub/. [Accessed 28 March 2021].

[13] K. G. a. W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Lviv, Ukraine, 2020.

[14] "What is NGINX?," NGINX, [Online]. Available: https://www.nginx.com/resources/glossary/nginx/. [Accessed 27 March 2021].

[15] "What is Ansible?," IBM Cloud, 24 November 2020. [Online]. Available: https://www.youtube.com/watch?v=fHO1X93e4WA. [Accessed 27 March 2021].

[16] "Develop a Web App Using Amazon ECS and AWS Cloud Development Kit (CDK)," AWS Online Tech Talks, 26 September 2019. [Online]. Available: https://www.youtube.com/watch?v=184S7ki6fJA. [Accessed 28 March 2021].

[17] "git --distributed-even-if-your-workflow-isnt," Software Freedom Conservancy, Inc., [Online]. Available: https://git-scm.com/. [Accessed 23 May 2021].

[18] "AWS CloudFormation," Amazon, [Online]. Available: https://aws.amazon.com/cloudformation/. [Accessed 28 March 2021].

[19] "AWS EC2," Amazon, [Online]. Available: https://aws.amazon.com/ec2/. [Accessed 28 March 2021].

[20] "AWS Elastic Container Service," Amazon, [Online]. Available: https://aws.amazon.com/ecs/. [Accessed 28 March 2021].

[21] "AWS Fargate," Amazon, [Online]. Available: https://aws.amazon.com/fargate/.

[Accessed 28 March 2021].

[22] "AWS Lambda," Amazon, [Online]. Available: https://aws.amazon.com/lambda/. [Accessed 28 March 2021].

[23] "AWS Simple Queue Service," Amazon, [Online]. Available: https://aws.amazon.com/sqs/. [Accessed 28 March 2021].

[24] "AWS Simple Notification Service," Amazon, [Online]. Available: https://aws.amazon.com/sns/. [Accessed 28 March 2021].

[25] "AWS S3," Amazon, [Online]. Available: https://aws.amazon.com/s3/. [Accessed 28 March 2021].

[26] "AWS Relational Database Service," Amazon, [Online]. Available: https://aws.amazon.com/rds/. [Accessed 28 March 2021].

[27] "AWS DynamoDB," Amazon, [Online]. Available: https://aws.amazon.com/dynamodb/. [Accessed 28 March 2021].

[28] "AWS Elastic Container Registry," Amazon, [Online]. Available: https://aws.amazon.com/ecr/. [Accessed 28 March 2021].

[29] "AWS Application Load Balancer," Amazon, [Online]. Available: https://aws.amazon.com/elasticloadbalancing/application-load-balancer/. [Accessed 28 March 2021].

[30] "AWS Secrets Manager," Amazon, [Online]. Available: https://aws.amazon.com/secrets-manager/. [Accessed 28 March 2021].

[31] "AWS Virtual Private Cloud," Amazon, [Online]. Available: https://aws.amazon.com/vpc/. [Accessed 28 March 2021].

[32] "AWS Secrets Manager," Amazon, [Online]. Available: https://aws.amazon.com/secrets-manager/. [Accessed 28 March 2021].

[33] S. Lauesen, Software Requirements: Styles and Techniques, Pearson Education Limited, 2002.

[34] "About Node.js," OpenJS Foundation, [Online]. Available: https://nodejs.org/en/about/. [Accessed 4 April 2021].

[35] "This is My Architecture," Amazon, [Online]. Available: https://aws.amazon.com/architecture/this-is-my-architecture/?tma.sort-by=item.additionalFields.airDate&tma.sort-order=desc. [Accessed 1 April 2021].

[36] J. Backes, S. Bayless, B. Cook, B. Kocik, C. Dodge, A. Gacek, S. McLaughlin, A. Hu, T. Kahsai, E. Kotelnikov, J. Kukovec, J. Reed, N. Rungta, J. Sizemore, M. Stalzer, P. Srinivasan and Suboti, "Reachability Analysis for AWS-Based Networks," in *31st International Conference on Computer Aided Verification, CAV 2019*, New York, 2019.

[37] "Service load balancing," Amazon, [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-load-balancing.html. [Accessed 3 April 2021].

[38] D. Gannon, R. Barga and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing,* vol. 4, no. 5, pp. 16-21, 2017.

[39] M. Kiran, P. Murphy, I. Monga, J. Dugan and S. Singh Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *IEEE International Conference on Big Data*, Santa Clara, 2015.

[40] H. Kniberg and M. Skarin, Kanban and Scrum: making the most of both, C4Media, 2010.

[41] "Trello," Atlassian, [Online]. Available: https://trello.com/en. [Accessed 24 May 2021].

[42] "NodeJS Downloads," OpenJS Foundation, [Online]. Available:

https://nodejs.org/en/download/. [Accessed 4 April 2021].

[43] "Docker Desktop," Docker, [Online]. Available: https://www.docker.com/products/docker-desktop. [Accessed 4 April 2021].

[44] "AWS Command Line Interface," Amazon, [Online]. Available: https://aws.amazon.com/cli/. [Accessed 4 April 2021].

[45] "AWS Account and Access Keys," Amazon, [Online]. Available: https://docs.aws.amazon.com/powershell/latest/userguide/pstools-appendix-sign-up.html. [Accessed 4 April 2021].

[46] "ExpressJS," OpenJS Foundation, [Online]. Available: https://expressjs.com/. [Accessed 4 April 2021].

[47] "Using middleware," OpenJS Foundation, [Online]. Available: https://expressjs.com/en/guide/using-middleware.html. [Accessed 16 April 2021].

[48] "Using the Amazon SQS message deduplication ID," Amazon, [Online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/using-messagededuplicationid-property.html. [Accessed 18 April 2021].

[49] G. Balasubramanian, "Choosing the Right DynamoDB Partition Key," Amazon, [Online]. Available: https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/. [Accessed 18 April 2021].

[50] "Using Global Secondary Indexes in DynamoDB," Amazon, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html. [Accessed 18 April 2021].

[51] "AWS EventBridge," Amazon, [Online]. Available: https://aws.amazon.com/eventbridge/. [Accessed 18 April 2021].

[52] "CDK Construct library for higher-level ECS Constructs," Amazon, [Online]. Available: https://docs.aws.amazon.com/cdk/api/latest/docs/aws-ecs-patterns-readme.html. [Accessed 1 May 2021].

[53] M. Gegick and S. Barnum, "Least Privilege," Department of Homeland Security, [Online]. Available: https://us-cert.cisa.gov/bsi/articles/knowledge/principles/least-privilege. [Accessed 18 April 2021].

[54] "time command in Linux with examples," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/time-command-in-linux-with-examples/. [Accessed 23 April 2021].

[55] "AutoCannon," NPM, [Online]. Available: https://www.npmjs.com/package/autocannon. [Accessed 24 April 2021].

[56] "Rolling Deployments," Amazon, [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/rolling-deployments.html. [Accessed 25 April 2021].

8 Appendix

8.1 HTTP benchmarking script

```
const autocannon = require('autocannon')
const fs = require('fs')

const URL = 'http://CdkSt-HttpS-4U6BNQMBMJDH-843218836.eu-west-
1.elb.amazonaws.com/create'
const DURATION = 360
const BUCKET_SIZE = 5000
const NUM_BUCKETS = DURATION * 1000 / BUCKET_SIZE
const STATS_PATH = __dirname + '/stats.json'
const CSV_PATH = __dirname + '/file.csv'

// Each spec is the arguments to be passed to runTest()
const specs = [
  ['DURING_DEPLOY', 10],
]

let collection
try {
  collection = JSON.parse(fs.readFileSync(STATS_PATH))
} catch {
  collection = []
}

const runTest = async (label, connections = 1) => {
  const start = Date.now()
  const test = {
    label,
    start,
    requests: [],
  }

  const result = await autocannon({
    url: URL,
    connections,
    duration: DURATION,
    requests: [
      {
        onResponse: (status, body) => {
          console.log(body)
          if (status === 200) {
            test.requests.push(Date.now())
          }
        }
      }
    ]
  })
  test.result = result
  collection.push(test)
}

const createBuckets = (test) => {
  const { requests, start } = test
  const normalized = requests.map(req => req - start)

  const histogram = new Map()
  for (const val of normalized) {
    const key = Math.floor(val / BUCKET_SIZE)
    if (!histogram.has(key)) {
      histogram.set(key, 0)
    }
    histogram.set(key, histogram.get(key) + 1)
  }
  test.buckets = histogram
}

const createCSV = (tests) => {
  let str = ['bucket', ...tests.map(t => t.label)].join(',') + '\n'

  for (let i = 0; i < NUM_BUCKETS; i++) {
    str += [i, ...tests.map(t => t.buckets.get(i) || 0)].join(',') + '\n'
  }

  fs.writeFileSync(CSV_PATH, str)
}

;(async () => {
  for (const spec of specs) {
    await runTest(...spec)
  }

  fs.writeFileSync(STATS_PATH, JSON.stringify(collection, null, 2))

  collection.forEach(createBuckets)
  createCSV(collection)
})()
```

8.2 Infrastructure code

```
const ecs = require('@aws-cdk/aws-ecs')
const cdk = require('@aws-cdk/core')
const ec2 = require('@aws-cdk/aws-ec2')
const ecsPatterns = require('@aws-cdk/aws-ecs-patterns')
const sns = require('@aws-cdk/aws-sns')
const sqs = require('@aws-cdk/aws-sqs')
const dynamodb = require('@aws-cdk/aws-dynamodb')
const lambda = require('@aws-cdk/aws-lambda')
const s3 = require('@aws-cdk/aws-s3')
const subscriptions = require('@aws-cdk/aws-sns-subscriptions')
const { SqsEventSource } = require('@aws-cdk/aws-lambda-event-sources')
const events = require('@aws-cdk/aws-events')
const targets = require('@aws-cdk/aws-events-targets')

class CdkStack extends cdk.Stack {
  constructor (scope, id, props) {
    super(scope, id, props)

    const topic = new sns.Topic(this, 'CreatedTopic', {
      contentBasedDeduplication: true,
      fifo: true,
      topicName: 'CreatedEntities'
    })

    const queue = new sqs.Queue(this, 'CreatedQueue', {
      contentBasedDeduplication: true,
      fifo: true,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      retentionPeriod: cdk.Duration.days(14)
    })

    // Automatically put published messages on the queue
    topic.addSubscription(new subscriptions.SqsSubscription(queue))

    const table = new dynamodb.Table(this, 'Entities', {
      partitionKey: { name: 'id', type: dynamodb.AttributeType.STRING },
      removalPolicy: cdk.RemovalPolicy.DESTROY
    })

    const indexName = 'stage-created-index'
    table.addGlobalSecondaryIndex({
      indexName,
      partitionKey: { name: 'stage', type: dynamodb.AttributeType.STRING },
      sortKey: { name: 'createdAt', type: dynamodb.AttributeType.NUMBER }
    })

    const bucket = new s3.Bucket(this, 'EntityBucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true,
      publicReadAccess: false
    })

    const vpc = new ec2.Vpc(this, 'Vpc')
    const albService = new ecsPatterns.ApplicationLoadBalancedFargateService(this,
'HttpService', {
      taskImageOptions: {
        image: ecs.ContainerImage.fromAsset('./services/simple-http-service'),
        environment: {
```

```
      SNS_ARN: topic.topicArn,
      TABLE_NAME: table.tableName
    }
  },
  cpu: 256,
  memoryLimitMiB: 512,
  desiredCount: 2,
  vpc
})

const scalableTarget = albService.service.autoScaleTaskCount({
  minCapacity: 2,
  maxCapacity: 10,
})

scalableTarget.scaleOnCpuUtilization('CpuScaling', {
  targetUtilizationPercent: 70,
  scaleInCooldown: cdk.Duration.seconds(60),
  scaleOutCooldown: cdk.Duration.seconds(60),
})

albService.targetGroup.configureHealthCheck({
  path: "/health",
  unhealthyThresholdCount: 10,
  interval: cdk.Duration.seconds(120),
  timeout: cdk.Duration.seconds(60),
})

const queueConsumer = new lambda.Function(this, 'QueueConsumer', {
  runtime: lambda.Runtime.NODEJS_14_X,
  handler: 'function.handler',
  code: lambda.Code.fromAsset('./services/queue-consumer'),
  environment: {
    S3_BUCKET: bucket.bucketName,
    TABLE_NAME: table.tableName
  },
  vpc,
  memorySize: 128
})

// Register queue consumer to process queue
queueConsumer.addEventSource(new SqsEventSource(queue, {
  batchSize: 10
}))

const task = new lambda.Function(this, 'Task', {
  runtime: lambda.Runtime.NODEJS_14_X,
  handler: 'function.handler',
  code: lambda.Code.fromAsset('./services/task'),
  environment: {
        S3_BUCKET: bucket.bucketName,
        TABLE_NAME: table.tableName,
        GSI_INDEX: indexName
      },
      vpc,
      memorySize: 128
    })

    const rule = new events.Rule(this, 'Rule', {
      schedule: events.Schedule.expression('rate(2 minutes)')
    })
    rule.addTarget(new targets.LambdaFunction(task))

    // HTTP service permissions
    topic.grantPublish(albService.taskDefinition.taskRole)
    table.grantWriteData(albService.taskDefinition.taskRole)

    // Consumer function permissions
    bucket.grantPut(queueConsumer.role)
    table.grantWriteData(queueConsumer.role)

    // Task function permissions
    bucket.grantDelete(task.role)
    table.grantReadWriteData(task.role)
  }
}

module.exports = { CdkStack }
```

8.3 HTTP service code

```
const express = require('express')
const os = require('os')
const AWS = require('aws-sdk')
const { uuid } = require('uuidv4')

AWS.config.update({ region: 'eu-west-1' })

const app = express()
const dbClient = new AWS.DynamoDB.DocumentClient()
const snsClient = new AWS.SNS()

// Introduce artificial slowdown to simulate real business logic
app.use((req, res, next) => {
  for (let i = 0; i < 1e9; i++) {}
  next()
})

app.get('/create', async (req, res) => {
  try {
    const name = req.query.name
    const id = uuid()

    await snsClient.publish({
      Message: id,
      TopicArn: process.env.SNS_ARN,
      MessageGroupId: 'NAME_GROUP'
    }).promise()

    await dbClient.put({
      TableName: process.env.TABLE_NAME,
      Item: {
        id,
        name,
        createdAt: Date.now(),
        processedAt: null,
        deletedAt: null,
        stage: 'created'
      }
    }).promise()

    res.status(200).json({ message: `Successful created on host: ${os.hostname()}` })
  } catch (err) {
    console.log(err)
    res.status(500).json({ message: 'Internal server error, check logs for details' })
  }
})

app.listen(process.env.PORT, () => { console.log(`App listening on port
${process.env.PORT}`) })
```

8.4 Consumer function code

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dbClient = new AWS.DynamoDB.DocumentClient()

exports.handler = async (event) => {
  for (const record of event.Records) {
    const id = JSON.parse(record.body).Message
    await s3.putObject({
      Body: id,
      ContentType: 'text/html',
      Bucket: process.env.S3_BUCKET,
      Key: `${id}.txt`
    }).promise()

    await dbClient.update({
      TableName: process.env.TABLE_NAME,
      Key: {
```

```
      id
    },
    UpdateExpression: 'set processedAt = :p, stage = :s',
    ExpressionAttributeValues: {
      ':p': Date.now(),
      ':s': 'processed'
    },
    ReturnValues: 'NONE'
  }).promise()
 }
 return {}
}
```

8.5 Recurring task function code

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dbClient = new AWS.DynamoDB.DocumentClient()

exports.handler = async () => {
 const data = await dbClient.query({
  TableName: process.env.TABLE_NAME,
  IndexName: process.env.GSI_INDEX,
  ProjectionExpression: 'id',
  KeyConditionExpression: 'stage = :s and createdAt <= :d',
  ExpressionAttributeValues: {
   ':s': 'processed',
   ':d': Date.now() - 60 * 60 * 1000
  }
 }).promise()

 for (const { id } of data.Items) {
  await s3.deleteObject({
   Bucket: process.env.S3_BUCKET,
   Key: `${id}.txt`
  }).promise()

  await dbClient.update({
   TableName: process.env.TABLE_NAME,
   Key: {
    id
   },
   UpdateExpression: 'set deletedAt = :d, stage = :s',
   ExpressionAttributeValues: {
    ':d': Date.now(),
    ':s': 'deleted'
   },
   ReturnValues: 'NONE'
  }).promise()
 }
```

```
  return {}
}
```