Tampere University

Juho-Pekka Vasenius

Migrating manually configured AWS infrastructure to use IaC approach

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
June 2022

Abstract

Juho-Pekka Vasenius: Migrating manually configured AWS infrastructure to use IaC approach
Master's thesis
Tampere University
Master's Degree Programme in Computer Sciences
June 2022

Server infrastructure management is moving towards big cloud service providers. The complexity of the systems deployed in cloud is increasing and this calls for new ways to manage the resources. Infrastructure as Code (IaC) is a modern approach to infrastructure management. It leads the system administrators to think of the infrastructure the same way they think about software code. The same development practises and methods can apply to the cloud infrastructure.

This study dives into the definition and key underlying concepts of IaC. It looks into the history of cloud computing and the problems that IaC was evolved to solve.

Thesis has a practical case study part where a production environment on Amazon Web Services cloud was migrated to use IaC deployment system. Results of the case study are evaluated against a recent quality model implemented for analyzing cloud services. The challenges and benefits are reported from the point of view of a small development team looking to upgrade their cloud infrastructure deployment model to IaC. Analyze indicates positive results on a large amount of the quality dimensions. Benefits are especially noticeable when examining the effects IaC has on the accountability and maintainability of the system.

Keywords: Infrastructure as Code, Cloud computing, Amazon Web Services, AWS Cloud Development Kit

The originality of this thesis has been checked using the Turnitin Originality Check service.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Research Questions 1
1.3 Scope and delimitations 2
1.4 Structure of the thesis 2
2 Cloud Computing 4
2.1 History of cloud computing 4
2.2 Relevant components of cloud infrastructure 5
2.2.1 Compute resources 5
2.2.2 Storage resources 6
2.2.3 Networking resources 7
2.3 Infrastructure Management 8
2.4 Issues on Infrastructure Management 8
3 Infrastructure as Code 9
3.1 Definition of IaC 9
3.1.1 Management operations 9
3.1.2 Principles and methods 9
3.1.3 Properties of the IaC environments 10
3.2 Declarative and Imperative systems 11
3.3 IaC scripting 12
3.4 IaC with Domain Specific Languages 13
3.5 IaC with general purpose languages 15
4 Quality Criteria 18
4.1 Accountability 18
4.2 Availability 18
4.3 Maintainability 19
4.4 Scalability and Elasticity 20
4.5 Performance 20
4.6 Security and Privacy 21
4.7 Usability 21
4.8 Reliability 21
4.9 Features and Functionality 22
4.10 Recoverability 22
4.11 Compliance 22
4.12 Empathy 23

5 Case study . . . 24
5.1 Background . . . 24
5.2 Requirements . . . 24
5.3 Implementation . . . 25
5.3.1 Mapping of the existing infrastructure . . . 25
5.3.2 Good practices . . . 25
5.3.3 Challenges . . . 26
5.3.4 Implementing the definition files . . . 27
5.4 Evaluation . . . 29
5.4.1 Accountability . . . 30
5.4.2 Availability . . . 30
5.4.3 Maintainability . . . 31
5.4.4 Scalability and Elasticity . . . 31
5.4.5 Performance . . . 32
5.4.6 Security and Privacy . . . 32
5.4.7 Usability . . . 33
5.4.8 Reliability . . . 34
5.4.9 Features and Functionality . . . 34
5.4.10 Recoverability . . . 35
5.4.11 Compliance . . . 35
5.4.12 Empathy . . . 36
5.5 Results . . . 36
6 Conclusion . . . 39
References . . . 43

1 Introduction

This chapter will start by describing the motivation for the thesis in Section 1.1. The research question and their reasoning are explained in the following Section 1.2. The Section 1.3 addresses the scope and delimitations of the work and the final Section 1.4 is about the structure of the thesis.

1.1 Motivation

Setting up server environments used to require a significant amount of physical labor. In the days of ”Iron Age” you had to send a purchase order to a vendor, wait for shipping, assemble them and install necessary software. Forgetting to order any components would cause delay (Morris 2021).

With on-demand cloud computing platforms came the possibility of setting up computing infrastructure with interactive tools. Processing power is available in a moments notice. The speed of the deployment is not the only benefit of the ”Cloud Age”. Properties like pay-on-demand pricing and scalability greatly help in creation of cost-effective and flexible IT solutions (Breitenbücher et al. 2014).

Infrastructure as code (IaC) is a modern approach to setting up, configuring and managing cloud computing resources. Breitenbücher and others (2014) state that the automation of application provisioning is one of the most important issues in cloud computing. Modern IaC approach takes a step further and automates this process with a declarative deployment system using general purpose programming languages.

This study investigates the benefits of using an IaC approach and practically implementing it in a situation where a manually created production infrastructure already exists to find out if utilizing IaC with the latest trend tool like Amazon Web Services Cloud Development Kit (AWS CDK documentation 2022) can be useful for a startup business looking for updating its backend systems.

1.2 Research Questions

This thesis work seeks to answer the following questions:

- RQ1 What problems was IaC evolved to solve?
- RQ2 What are the major technologies and methodologies behind IaC?
- RQ3 What are the general effects IaC has on service quality?

- RQ4 What are the effects IaC has on service quality from startup point of view?

First two research questions seek to answer why and how this technology was developed to better understand its current state. The answer for them is mainly searched by the means of literary review utilizing academic articles, practical instructive books and manuals or relevant documentation available online.

Research questions three and four are about the consequences and effects that IaC implementations have on software service quality. Answers for the research questions three and four are to be answered partly by reflecting with the literature and partly with the practical expertice achieved by implementing an IaC solution to replace a manually configured environment in an actual business environment.

1.3 Scope and delimitations

This work aims to deliver a comprehensive review to the current state of IaC practices. The practical point of view is from a small business stakeholder looking to invest into upgrading their systems with this technology. This emphasizes the following questions: What are the major implementation options? What does the minimal viable product (MVP) look like with IaC? What are the challenges of adopting IaC practises?

The practical case study does not implement a perfect IaC solution following all the advanced practices and methods like continuous infrastructure deployment or automated infrastructure testing. It does however implement a comprehensive IaC deployment system for a non trivial production environment serving actual customers.

The word startup is used in the research questions to highlight the effects which limited resources can have. Term is commonly used about newly established businesses working on tech industry.

1.4 Structure of the thesis

The next chapter covers cloud services in general. It seeks to build a basis for understanding of how cloud services have evolved to their current state and to situate the IaC practises into them. The third chapter covers the definition and the core practises behind IaC. It also has practical examples of how different styles of infrastructure deployments are handled in AWS environments. Fourth chapter introduces a quality model suitable for analyzing IaC. Each dimension of the quality model is visited and reviewed. Fifth chapter is about the practical case study. The case study implemented an IaC solution for a Finnish software company. Results of

the implementation are contemplated against the quality dimensions of the quality model. The final chapter is about the conclusion and future work.

2 Cloud Computing

This chapter introduces the cloud services in general. Section 2.1 addresses the history of cloud computing followed Section 2.2 and subsections exploring the parts of cloud infrastructure most relevant for the IaC and this study. Section 2.3 addresses the infrastructure management and the final Section 2.4 examines the common issues and challenges of cloud infrastructure management.

2.1 History of cloud computing

Cloud computing is a model which delivers an on-demand connection to computing resources. They typically include networking, storage and computing resources delivered as different kind of services. These resources can be provisioned fast and their management is highly automated so that the service provider has minimal to none manual interaction (Ruparelia 2006).

Early implementations resembling cloud computing were founded already during the 1960s. This was the era of the large mainframe computers accessed by remote workstations utilizing a concept called time-sharing. It had the benefit of a computer to be used simultaneously by two or more people. (Foote 2021)

Term "virtualization" was also used to for this process, but the meaning of the word has drifted since. The term came back in the 1990s and is now more commonly used when talking about *virtual machines*, emulations of computer systems which are run as guests on host computers. Kumara and others (2021) state that the invention of virtualization was one of the prerequisites for modern cloud computing.

The use of virtual computers became popular in the 1990s when telecommunication companies started offering virtual private networks (VPN) as rentable services. This launched the use of virtual computers and lead to the development of the modern cloud computing infrastructure which offers the option to replace up-front investments on server infrastructure with lower costs that scale with the business.

In 2006 Amazon started offering cloud services in the form of Amazon Web Services (AWS). Google Cloud Platform was released in 2008 providing services including data analytics, data storage, computing and machine learning. In 2010, Microsoft released its counterpart, Microsoft Azure. It has similar services some of them also being Microsoft-specific (Amazon Web Services documentation 2022; Google Cloud Platform documentation 2022; Microsoft Azure documentation 2022).

The three service providers control 65% of the market. According to estimates of Synergy Research Group, AWS leads the market with a worldwide market share at 33%, Microsoft is the second at about 22% and Google Cloud is the third at 10%.

The trailing 12 months revenues for all cloud infrastructure services including the first quarter of 2022 are estimated to reach 191 billion dollars (Synergy Research Group 2022).

2.2 Relevant components of cloud infrastructure

Modern cloud platforms include many parts, but in the very basic form, the cloud infrastructure platforms offer storage, computing and networking capabilities. Morris (2021) suggests a hierarchical grouping to organize the parts into three categories: applications, application runtimes and infrastructure platform as expressed in the Table 1.

All the components in the infrastructure exist to enable the *applications*, the software, that organizations and private users can utilize. *Application runtimes* are what gives resources and services for the individual applications. They contain constructs like application servers, databases and container clusters. This category can also be described as Platform as a Service (PaaS). The third group is the *infrastructure platform*. It contains infrastructure resources like computing, storage and networking primitives and their management tools. The following subsections will examine each type of the infrastructure resources in more detail.

Application Packages	Container Instances	Serverless Code
Applications		

Servers	Container Clusters	Database Clusters
Application Runtime Platforms		

Compute Resources	Storage	Network Resources
Infrastructure Platforms		

Table 1 *Layers of cloud system elements.*

2.2.1 Compute resources

Compute resources are about executing program code on a CPU and there are few different ways of delivering this power. Compute resources include: virtual machines, physical servers, server clusters, Container as a Service (CaaS) containers, application hosting clusters and Function as a Service (FaaS) serverless code runtimes.

Virtual machines are the basic building blocks of cloud services. They are full emulations of operating systems which run as guests on a host computer and are given a virtualized access to hardware resources. Important hardware resource categories are processing, memory and input/ouput resources. These kind of virtual

machines are also called *System virtual machines* to differentiate them from *Process virtual machines* which are used to run a single program in an isolated environment. Utilizing virtual machines is beneficial for cloud service providers as many virtual machines can be run on a single physical device. This is especially useful for performance efficiency, since it is common for the hardware resources of a typical computer system to be underutilized. Even though the virtual machines share the same hardware resources, the data within them is still highly secured (Craig 2006).

Besides virtual servers, cloud vendors also offer physical servers. Physical servers run a single operating system and their main benefit is the performance, as the hardware is accessed directly by the operating system instead through the virtualization layer.

Server clusters are managed systems where multiple server instances are used with a load balancer to achieve more capacity or reliability. Load balancer receives requests for work and distributes them to worker instances. Amount of instances in the cluster can be scaled to match the load. Typical setup includes minimum and maximum number parameters for the instance count as well as rules for when to increment or decrement the number of running instances. The rules might be a predefined schedule or dynamic so that they react to the current load of the system.

Many public cloud infrastructure platforms offer higher level services for deploying and managing applications. They bundle different resources together to give user an easier configuration experience and reduce management complexity. Examples for these hosted Platform as a Service (PaaS) services are AWS Elastic Beanstalk, Heroku and Azure Devops (Morris 2021).

A relatively new concept are the Function as a Service (FaaS) "serverless" applications. They give the developer a very fast way to deploy applications, sometimes as small as a single function with minimal server configuration. The configuration can be as simple as selecting which programming language to use. The servers running this computing are not started or controlled directly. Instead each execution is run in a new environment and this makes the instances stateless. All input and output data needs to be stored externally (AWS Lambda documentation 2022).

2.2.2 Storage resources

Storage resources include block storage also known as virtual disk volumes, object storage, networked filesystems, structured data storage and secrets management.

Block storage can be used to implement disk volumes for servers. The disk volumes are virtual, they don't have to be part of the same physical device, rather the block storages allocate memory from a larger pool of resources, common to all the servers.

Object storage offer an easy way to store individual pieces of data with minimal

initial configuration. Vassallo (2020) instructs the developers to at least consider using AWS S3 objects storage service when storing almost any data. Benefits of using objects storage are low costs, pay-on-demand pricing and practically unlimited capacity.

Networked filesystems can be mounted on many host computers. Shared drives are useful for storing and granting file access to team members. They can also be used by servers for example to combine log files or to provide access to common configurations and software packages.

Structured data storage is data which is stored in predefined format. This means most importantly different relational databases such as MySQL, PostgreSQL and SQL server. Structured data storage might also be just key-value strings or some other data structures or definition languages such as JSON or XML files. Closely attached to structured data storage, the cloud vendors provide tools for processing the data.

Secrets management is a special case of storage which is designed to handle different kind of credentials and keys needed for security and privacy of the cloud.

2.2.3 Networking resources

Cloud resources need rules for communicating with each other and the rest of the world. Cloud vendors provide tooling for this in the form of networking resources: Network address blocks, Network access rules (firewall), Asynchronous messaging, DNS entries, Load balancing rules, VPNs, Routes, Gateways, Proxies, API gateways and Cache.

Describing details for all of these is outside the scope of this study, but those which are critical for the case study are covered.

Amazon Web Services calls its top-level network address block Virtual Private Cloud (VPC). Top-level block can be divided into subnets and those can be associated with locations like *Central EU* or *US East 1* and they actually describe the physical locations of the server rooms.

Deciding rules for the communication between the resources is done with network access rules. They govern which IP addresses or DNS names can access certain ports in the network. In AWS they can be described in constructs called *Security Groups* which can then be associated with other resources such as a database server.

Another networking service on AWS is the *Route 53* which bundles all the tools necessary for managing Domain Name System (DNS) records.

2.3 Infrastructure Management

Cloud infrastructure management consists of many tasks. In general these tasks are about developing, maintaining and troubleshooting the cloud infrastructure. The areas to which these tasks are targeted are typically the network connections, service deployments, installations, integrations and service deployments. Systems need to be kept updated to maintain a necessary level of features and security. This is done by applying patches and upgrades. Management tasks also include handling backups of critical data, setting up necessary monitoring and documenting the environments (Morris 2021).

2.4 Issues on Infrastructure Management

Configuration drift is variation which happens over time to environments that once were identical. Systems that need to serve a large amount of requests typically have some degree of repetition. For example this repetition can come in a form of multiple web servers, multiple databases or multiple firewall settings. The environments can be considered to be identical if their consequential configuration is identical. They naturally handle different data and so the environment drift is caused by differences in the configuration, not the differences in the data. Environment drift can be caused by the operators themselves. System administrator might change the configuration of a single environment for troubleshooting or for a maintenance task, but leave other similar environments untouched. Such an environment is sometimes called a *snowflake* to emphasize its uniqueness and perhaps also fragility (Morris 2021).

Managing costs with cloud services is not always easy. Cloud services do provide on-demand pricing, but implementing systems which use this possibility effectively is not necessarily easy. For example developers usually require some kind of testing environment to work with. This testing environment should reflect the production environment as closely as possible to be able to reproduce loads and test cases that the production environment will experience. Setting up these environments can multiply the costs of the cloud service. Special expertice is required to make the cloud infrastructure scalable.

Data security is the shared responsibility of the cloud administrator and the service provider. Cloud service providers expect the administrators to implement necessary good practises related to the data security and to be educated in the subject. Additional security counseling can be bought, but is not included in the regular subscription plans.

3 Infrastructure as Code

This chapter will focus on Infrastructure as Code by looking into the suggested definitions of it and by describing the core technologies behind it. First Section 3.1 will cover the different suggested definitions for the practice. Following that in Section 3.2 there is a comparison and definitions for two major deployment system models utilized by IaC. Sections 3.4 - 3.6 describe the different types of definition files and scripts used to implement IaC.

3.1 Definition of IaC

Kumara and others (2021) propose the following three major dimensions to define Infrastructure as Code: management operations, principles and methods and properties of the IaC environments. They are covered in the following subsections.

3.1.1 Management operations

The first dimension is the types of management operations supported by IaC. The management operations target the infrastructure, software/platforms and application-specific capabilities of the computing environment. They can be divided into three groups: infrastructure templating, configuration management and configuration orchestration.

Configurations orchestration provides the infrastructure resources. Infrastructure templating consists of generating a catalog of images for necessary infrastructure resources such as containers and virtual machines. Configuration management is used run installations on the provisioned servers and manage their configuration (Kumara et al. 2021).

3.1.2 Principles and methods

The second dimension is the principles and methods of realizing the management operations. Instead of using conventional tools like interactive shells or GUI consoles, the infrastructure is defined with domain specific languages (DSL). There are two main models for the IaC languages: declarative and imperative / procedural. These models are discussed in more detail in the Section 3.2.

Hoban (2022) states that IaC brings software engineering methods and discipline to the management of computing environments. Kumara and others (2021) claim that *any* software developing methodologies can be used with IaC. They say that many practitioners recommend adapting *DevOps* and continuous integration (DI) or continuous deployment (CD) methodologies for IaC. IaC does make it possible to

automate packaging, configuring and testing the infrastructure in a CI/CD pipeline. Morris (2021) points out that it is still somewhat rare to see teams fully implementing this.

3.1.3 Properties of the IaC environments

The third dimension is the desired properties of the managed environments. Kumara and others (2021) list six of these properties: virtualization, software-defined / programmable infrastructure, consistency, immutability, auditability and reproducibility. They also state that IaC environments often try to fulfill two key properties: idempotence and transparency.

Some properties are considered as prerequisites for IaC and others are induced by it. Prerequisites include virtualization and software-defined / programmable infrastructure. The concept of virtualization is visited in Subsection 2.2.1 and the concept of software-defined / programmable infrastructure is examined in the Section 3.3. Induced properties are immutability of the infrastructure, consistency among multiple environments, auditability and reproducibility (Kumara et al. 2021) (Morris 2021).

Idempotence is a property of a mathematical function where the result of a function is always the same for the given parameter, no matter how many times the function is applied. Hummer and others (2013) point out that rigorous testing is needed to ensure that the environment will change state idempotently to the desired state from arbitrary starting state.

Transparency refers to the visibility of the state of the system. IaC definition files provide both the documentation and the current state of the environment. Morris (2021) reminds that this should not be understood too literally, as the infrastructure definition files are still rarely the only documentation that is needed. Businesses often have stakeholder groups who lack the technical knowledge to read such files.

Immutability indicates that the infrastructure can not be modified, once it has been deployed. Instead, new infrastructure should be provisioned as it is easy to do with IaC methods.

Consistency among multiple environments together with reproducibility can be reached because the environments are created automatically from the same definition files. This can greatly lessen the possibility of human error.

Auditability refers to the possibility to follow the changes made to the environments by examining the version control where the definition files are stored. Also the deployment services typically store a log of all the deployments and data about who initiated them.

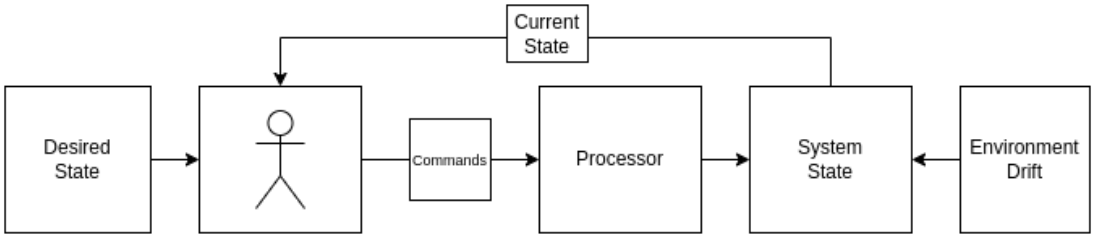

Figure 1 *Imperative deployment system.*

3.2 Declarative and Imperative systems

There are two main approaches for the IaC languages: declarative models and imperative / procedural models (Endres et al. 2017; Kumara et al. 2021). Both approaches have their benefits and drawbacks (Breitenbücher et al. 2014).

With imperative / procedural system, the user has to know the wanted state of the system and the exact commands needed to transition to the wanted state. Part of the system that handles the commands is called ”processor”. Unless the building of the system is started from the scratch, the user also has to know the initial state of the system. An imperative deployment system is described in Figure 1.

Declarative systems require a deployment and management system which can take a model describing the structure of the environment and come up with steps to transition the environment from an arbitrary state to the desired state. Declarative deployment system is therefore more automated than an imperative deployment system. With this system, the user only has to know the desired system state. The system then utilizes a component called a ”controller” to find out the necessary operations to reach the desired state. A declarative deployment system is described in Figure 2.

Different phases of the controllers work can be displayed and reported to the user. The controller can typically describe what are the differences of the desired state and the current state. This can be useful when reviewing the changes that are to be deployed to the system without actually deploying them. Some systems also allow the user to acquire the commands which are detected to be necessary for the desired state change.

Endres and others (2017) suggest that declarative deployment model pattern is better suited for simple applications which require only few or no customization, whereas imperative deployment models are needed to achieve more application-specific customization.

Knowing how to apply both the declarative and the imperative approach when implementing practical changes to the system seems to be valuable. Highly au-

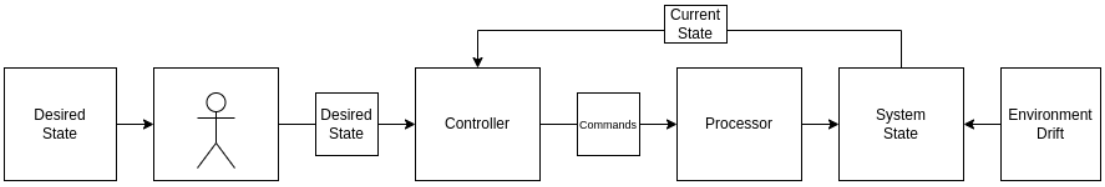

Figure 2 *Declarative deployment system.*

tomated systems can be more difficult to debug as they hide some of their inner workings from the user. If there is a problem with the system, it might be fastest to use imperative tools for the recovery, instead of deploying changes through the slower pipeline of the declarative system.

Endres and others (2017) point out that using imperative approach requires considerably more expertise as the developer has to be familiar with the cloud provider's application programming interfaces. We can recognize this as a danger when working with declarative systems: administrator might be able to create an environment which he/she does not fully understand. Short (2020) warns about a related issue when using a higher level declarative systems that "compile" down to Cloud Formation templates. Treating those templates as *assembly code* which developer don't have to understand about, might lead to worse decisions. The concept of CloudFormation templates is addressed in the sections below.

Endres and others (2017) also point out that some management tasks might still be only possible using this imperative approach. For the declarative deployment to work, the controller has to be able to transition the state of the system to the state defined in the definition files. This might be true for a high percentage of all the resources available in a specific cloud service, but different declarative systems are also used to implement finer details of server installations. For such situations it might be necessary to include some imperative scripting. Such obstacle could be for example an installer of a commandline application requiring manual choices inputted by the user during the installation process. Installation cannot be defined as a declarative representation as the controller would not yet understand it, but it could be automated procedurally.

3.3 IaC scripting

Before IaC systems, proficient server management was done with scripting. Languages like Bash, Perl, PowerShell, Ruby and Python are still essential tools for managing systems in cloud. Their usage is made possible by the software defined / programmable interfaces of the cloud services. Deploying cloud resources is a typical usecase for these interfaces, but they can be used for much more. In AWS

the functionality is equivalent to that provided by the browser-based management console. (Morris 2021).

Code Example 1 is using such interface with a Bash script. The script utilizes the AWS CLI command line interface to launch two virtual servers. They are set to be launched on the geographical availability zone ”eu-central-1” with an instance-type hardware setting ”t2.medium”, the count of intances to be deployed is 2 and the operating system image is defined as some specific id found in the catalog of choices available (AWS CLI documentation 2022).

```
1 #!/bin/bash
2 aws --region "eu-central-1" ec2 run-instances \
3 --instance-type t2.medium \
4 --count 2 \
5 --image -id ami-fa53bf454090ff0e5
```

Code Example 1 *Bash script leveraging AWS CLI.*

This kind of imperative / procedural scripting is useful and easy to replicate, but when working with a larger amount of infrastructure resources, it would be nicer to be able to write some kind of listing of the resources to a separate definition file and make the script create the resources based on it. However such system would not easily satisfy the requirement of idempotence. Each time we would run the script, more server instances would be created, even if we just wanted to change the instance-type. More logic would be required to first find out what kind of deployment has already been made.

3.4 IaC with Domain Specific Languages

CloudFormation (CFN) is the AWS service on which its native declarative IaC solutions are based. It was launched in 2011. It allows the operators to define stacks, collections of AWS resources, which can be deployed with a single command.

Stacks are defined in YAML or JSON markup languages. Markup languages differ from programming languages in that they are mostly used for describing static data-structures. They follow specific rules and formatting therefore creating a Domain Specific Language (DSL). Fowler and Parsons (2010) define a DSL as a ”small language, focused on a particular aspect of a software system”.

Code Example 2 is a simple CFN template which creates an object storage resource, with an *AccessControl* property set so that the content inside the bucket is publicly available. The template is written in YAML.

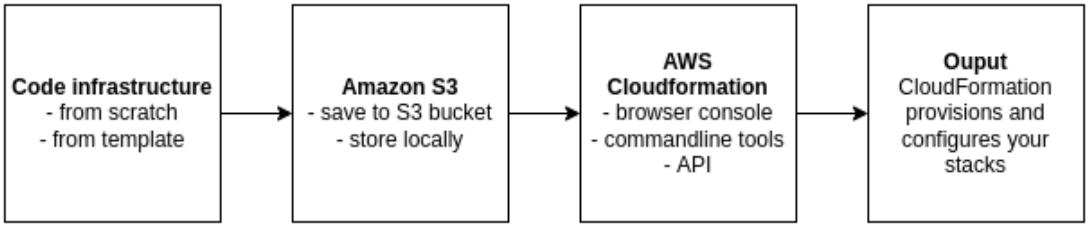

Figure 3 *CloudFormation flow chart.*

```
AWSTemplateFormatVersion: "2010-09-09"
Description: This is my first bucket
Resources:
    MyBucket:
        Type: AWS::S3::Bucket
        Properties:
            AccessControl: PublicRead
```

Code Example 2 *CloudFormation template.*

To utilize this template, the user needs to access the AWS CloudFormation service. It can be accessed in a few different ways as described in the Figure 3. User can either use the AWS Console web application to upload the template file or use the CloudFormation CLI as described in Code Example 3 . The AWS GUI console even has graphical designer tool with drag and drop functionality which automatically generates the CFN template code as shown in figure 4.

```
aws cloudformation create-stack \
    --stack -name my_stack \
    --template-body \
    MyBucket.yml
```

Code Example 3 *Bash Script leveraging CloudFormation CLI.*

The very basic features CloudFormation include: creating a new stack, updating a stack, deleting a stack. Little more advanced features include getting the outputs from the stack or creating stacks with roles. To implement management operations for multiple stacks, user would typically apply shell scripting. Code Example 4 is a Bash script which would delete three stacks.

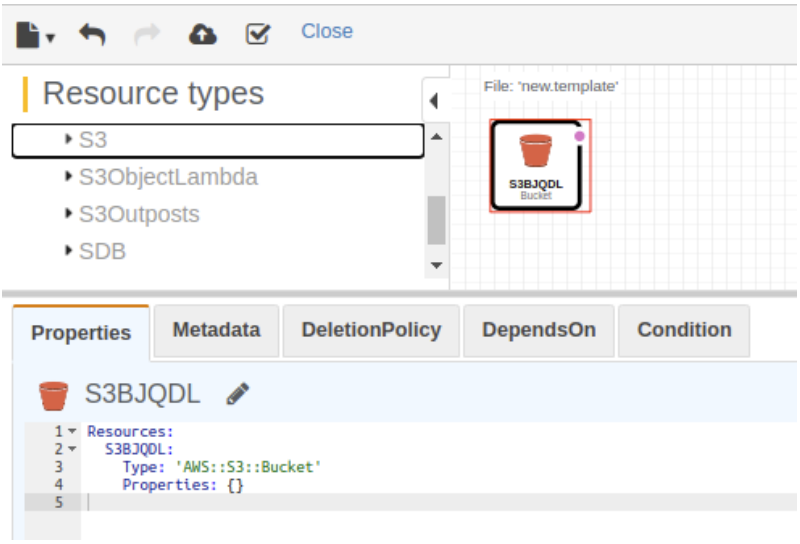

Figure 4 *AWS CloudFormation Designer tool.*

```
#!/bin/bash
for i in stack_one stack_two stack_three;
do aws cloudformation delete-stack --stack-name $i ; done
```

Code Example 4 *Deleting stacks with AWS CLI.*

Alternatives or similar systems to AWS CloudFormation include HashiCorp Terraform, AWS CloudFormation, Pulumi, Azure Resource Manager and Google Cloud Deployment Manager (Terraform documentation 2022; Pulumi documentation 2022; Azure Resource Manager documentation 2022; Google Cloud Deployment Manager documentation 2022).

3.5 IaC with general purpose languages

When talking about Infrastructure as Code, people can have varying definitions for the word *code*. Some might say that Domain Specific Languages (DSL) written in markdown languages like YAML, JSON or XML shouldn't even be called code, because they do not fulfill the requirement of being Turing complete (YAML documentation 2022; JSON documentation 2022; XML documentation 2022). Morris (2021) however argues against drawing such strong borders between the different types of IaC definitions files calling them all *code*.

AWS Cloud Development Kit implements Infrastructure as Code definition files with general purpose programming languages. The service was released in 2019 and Morris (2021) mentions it as being part of ”a new trend” of IaC. AWS CDK outputs CFN templates and implements the same management commands as CFN commandline interface.

Programming languages currently supported by the AWS CDK are: C#, Python, JavaScript, TypeScript, Java and Go. All the languages have their own documentation, and they all are promised to deliver the same features (AWS CDK documentation 2022).

Similarly to working with CloudFormation the developer has to define stacks of cloud resources. With AWS CDK this begins by writing a class definition which inherits from the core Stack class. To add resources to the stack, developer instantiates other classes of the CDK library inside the constructor function of the stack class. Classes within the AWS library represent all the cloud resources and settings that are available (AWS CDK Documentation 2022).

Before deploying the stacks, the definition files need to be processed with `cdk synth` command. This ”synthesize” step checks the validity of the code. This is done offline and the errors that can be identified at this point are syntax errors and exceptions that can be caused by invalid configurations.

After outputting the CFN templates user can choose to deploy the stack. The changes that the deployment would bring to the system can be examined before deployment with the `cdk diff` command, which compares the template code to an already deployed stack and lists the differences.

The Code example 5 is a minimal CDK Stack written in Python and the Code example 6 is a CloudFormation template that was synthesized from it. The stack contains a single AWS S3 object storage bucket. From the listings can be seen that the CFN template includes some default settings that were not implicitly defined in the python code. Therefore it is a good practise to always read through the synthesized CFN template.

```
from aws_cdk import aws_s3 as s3
from aws_cdk import core

class HelloCdkStack(core.Stack):

  def __init__(self, scope: core.Construct, constr_id: str,
               **kwargs) -> None:
    super().__init__(scope, constr_id, **kwargs)

    bucket = s3.Bucket(self, "MyStorageBucket")
```

Code Example 5 *AWS CDK Stack written in Python.*

```
{
  "Resources": {
    "MyFirstBucketB8884501": {
      "Type": "AWS::S3::Bucket",
      "UpdateReplacePolicy": "Retain",
      "DeletionPolicy": "Retain",
      "Metadata": {
        "aws:cdk:path": "HelloCdkStack/MyFirstBucket/Resource"
      }
    }
  }
}
```

Code Example 6 *CFN Template JSON synthesized from the source stack.*

Short (2020) mentions that it is dangerous to treat CloudFormation template files as the compiled assembly code of the CDK, which the users don't have to understand about. He calls for even a rudimentary understanding of CloudFormation for the developer to be able to make better decisions with higher level tools like AWS CDK.

4 Quality Criteria

Quality models can be used to compare, measure and represent the quality of services in a way that a collective understanding can be reached. Following sections cover the 12 dimensions of the ”Service Measurement Metrics Model” (S3MQual) proposed by Hourani and others (2019). The model is compatible for all the main cloud service types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). The model had its dimensions selected with following criteria: The relevance for the cloud computing services, the relevance for the business and customer needs and by the relevance to other quality models, especially those quality models which emphasized cloud quality. Lastly the general practicality and achievability was considered. Some of the dimensions can be given a quantitative description, while others should be regarded as qualitative dimensions (Hourani et al. 2019).

4.1 Accountability

Miguel and others (2014) describe accountability as ”The degree to which the actions of an entity can be traced uniquely to the entity or the degree to which events can be proven to have happened, so that they cannot be repudiated later”.

Accountability is the degree of which certain actions and changes can be tracked down to a specific actor. If a problematic breaking change was introduced to the software, the initiator of the change should be able to be found.

Accountability can also describe the degree to which problems are being fixed. Is the service provider working responsibly in relation to the customers in the moment of trouble? The dimension also describes the accountability of security exposures.

Garg and others (2013) summarize accountability by stating that organizations do not want to deploy their applications or store their data into places where there is not accountability of security exposures and compliance.

4.2 Availability

Miguel and others (2014) define availability as ”The degree to which a software component is operational and available when required for use”.

Availability can be presented in the following way:

$$\alpha = t/t_s \tag{4.1}$$

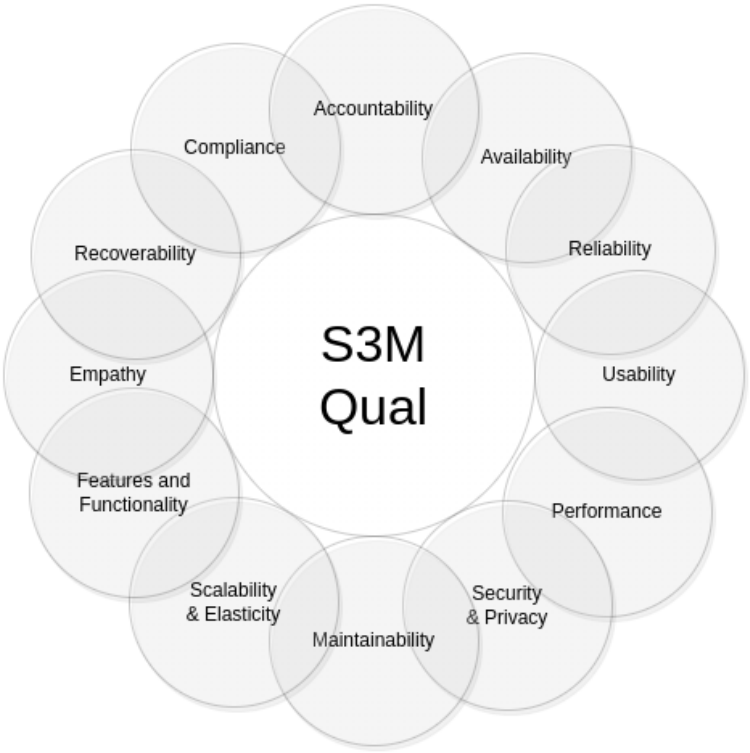

Figure 5 *S3MQual quality model dimensions.*

where $0 \leq \alpha \leq 1$ represents availability, t and t_s denote the uptime and the total time of an operational period. The closer α is to 1, the higher the availability.

Cloud services are provided through internet and customers expect them to be highly available. Ideally they should be working without interruptions (Zheng et al. 2014).

Service providers often provide some kind of availability goals and historical data of their availability. Availability requirements can be included in service contracts with possible penalties issued to the service provider in case they are not met.

4.3 Maintainability

Hourani and others (2019) describe maintainability as the degree to which the software product can be modified. These modifications may include improvements, corrections or adaptation of the software to changes in environment, requirements or functional specifications.

The degree to which a software product is able to fulfill these requirements is also called modifiability and another concept related to modifications is modification stability, the degree to which a software product can avoid problems caused by modifications. (Miguel et al. 2014).

Concept of modularity has a relation to maintainability. Modularity can be described as the degree to which a system is composed of individual discrete components such that allow a single component to be changed without compromising the function of other components (Miguel et al. 2014).

4.4 Scalability and Elasticity

Scalability and Elasticity define the ease with which an application or component can be modified to expand its existing capabilities. It includes the ability to accommodate major volumes of data.

Zheng and others (2014) suggest a formula for evaluating the elasticity of a cloud service:

$$\varepsilon = \frac{\sum_{i_1=1}^{n} r_{i_1}}{\sum_{i_2=1}^{n} r_{i_2}} \tag{4.2}$$

where $0 \leq \varepsilon \leq 1$ represents elasticity, r_{i_1} and r_{i_2} denote the amount of resources allocated, and the resources requested in the i:th request and n represents the number of requests in a time period. The closer ε is to 1, the higher is the elasticity.

We might add a practical limitation to the formula, by stating r_{i_1} should never be more than r_{i_2}. Otherwise, moments when the amount of allocated resources exceeds the amount of requested resources can cancel out the opposite situations and could lead to the score of $\varepsilon = 1$ even when the resources were insufficient at some point.

4.5 Performance

Performance measures the effectivity with which the service can be delivered.

One way to measure performance is to measure responsiveness. It measures the promptness with which the service answer to a request during a time interval. Zheng and others (2014) provide a formula for measuring responsiveness:

$$\tau = 1 - \frac{f_{i=1}^{n}(t_i)}{t_{max}} \tag{4.3}$$

where $0 \leq \tau \leq 1$ represents responsiveness, t_i denotes the time between the submission and the completion of the i:th request, t_{max} represents the maximum acceptable time to complete a request, n is the total number of requests and f is a

function which measures the central tendency of a set of data, such as the mean or the median.

Performance efficiency is the degree to which the service provides appropriate performance in relation to the amount of resources used (Miguel et al. 2014).

4.6 Security and Privacy

Hourani and others (2019) describe this dimension as "The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization". The key attributes of security are integrity, confidentiality and immunity.

Integrity can be defined as the degree to which the accuracy and completeness of assets are safeguarded (ISO-25010, 2011).

Confidentiality can be defined as the degree to which the software product provides protection from unauthorized disclosure of data (ISO-25010, 2011).

Immunity is the degree to which a product or system is resistant to attack (ISO-25010, 2011).

4.7 Usability

Usability dimension covers following attributes: installability, learnability, accessibility and operability (Hourani et al. 2019).

Zheng and others (2014) describe usability as how efficient, easy, and enjoyable the interface is to use, or assesses the ease of invocation if the functionality is delivered as an Application Programming Interface (API)."

Installability measures the degree to which a software product can be successfully installed and uninstalled in a specific environment (Miguel et al. 2014).

Learnability measures the degree to which a software product enables the learning of the use of the software product (Miguel et al. 2014).

Zheng and others (2014) point out that end users who have less experience with cloud services, have easier time understanding a Graphical User Interface (GUI) and is even better if it is delivered as a Web User Interface (WUI), which does not need installation. They also state that usability should be considered as a subjective dimension, because it is difficult to be given a quantitative description.

4.8 Reliability

Reliability is the degree to which a service can operate without failures.

Reliability can be measured by:

$$\rho = 1 - \frac{n}{n_s} \tag{4.4}$$

where $0 \leq \rho \leq 1$ represents reliability and n and n_s denote the number of failures and the total number of operations that occurred during a time interval. The closer p is to 1, the higher the reliability of the system is (Zheng et al. 2014).

4.9 Features and Functionality

Features and Functionality examines the degree to which the software product provides functions (Hourani et al. 2019). This dimension is purely quantitative, as it is difficult to evaluate if a poorly working feature gives a net positive impact for the service when compared to not having that feature at all.

Functional properties define what is offered. Cloud service providers typically provide computing, networking and storage services. The exact catalog of services differs between service providers. (Zheng et al. 2014).

4.10 Recoverability

Hourani and others (2019) describe recoverability as the degree to which the software product can re-establish a specified level of performance and avoid data loss in case of a failure.

After a failure, a computer system will sometimes be down for a period of time, the length of which is determined by its recoverability (ISO-25010, 2011).

Survivability is a concept related to recoverability and security. It covers the degree to which a product or system is able to continue to fulfil its mission in spite of the presence of attacks (ISO-25010, 2011).

4.11 Compliance

Compliance dimension examines the degree to which the software product adheres to standards, conventions, style guides and regulations (ISO-25010, 2011).

Ideally the team should consider necessary compliance mandates at an early phase of the project planning. Following necessary data security conventions are paramount for services handling sensitive customer information. On the lighter side of the compliance requirements there are source code style requirements which can be said to increase the quality of the source code.

Benefits of compliance can come from reduced time fixing code, increased visibility into the software security and reduced time spent for gathering documentation needed for security auditions.

4.12 Empathy

This dimension examines the degree to which a service adapts to the needs of an individual customer. Hourani and others (2019) describe the dimension further by stating "This dimension is about caring and providing attention to customers individually."

Berry and others (1988) describe empathy dimensions as "caring, individual attention the firm provides to its customers."

Miguel and others (2014) talk about the concept of helpfulness, which is the degree to which the software product provides help when user needs assistance.

5 Case study

This chapter is about an empirical case study, which implemented a migration for existing AWS infrastructure to use IaC approach. Chapter will start by describing the background of the project in the Section 5.1. The project requirements are described in the Section 5.2. Following Section 5.3 is the description of the implementation and the final Section 5.4 applies qualitative analysis to investigate the research questions. Quality analysis is based on a cloud system quality model "Service Measurement Metrics Model" proposed by Hourani et al. (2019).

5.1 Background

The case study project was done for a Finnish company producing a Point of Sales software. The system has been in production from late 2017 with over 6 million unique sales transactions made with it so far. The development for the IaC project was done by the author of this thesis with an initial support by a technical consult. Nobody in the team inside the company had significant experience in cloud service administration prior to the project.

The case example implements an IaC solution using Amazon Cloud Development Kit (Amazon CDK) on Amazon Web Services IaaS platform. The IaC solution was made to replace a manually created environment which had accumulated technical dept in the form of outdated versions and lacking documentation. AWS services used in the system are listed in the Table 2

Service	**Usage**
AWS Elastic Beanstalk	Application servers platform
AWS RDS	Database service
AWS S3	Object storage
AWS CloudFront	Content delivery network
AWS IAM	Identity and Access Management
AWS Route 53	Domain Name System service
AWS Cloud Development Kit	IaC development framework
AWS CloudFormation	IaC service

Table 2 *AWS services used in the case study.*

5.2 Requirements

The urgency for implementing the infrastructure updates came from Amazon retiring versions of the platforms we were using. The retiring versions won't receive

automatic updates which brings security risks to the cloud infrastructure. The developers who had originally created the cloud infrastructure, were no longer available and the documentation for the infrastructure was not well maintained. An external consultant suggested the IaC approach and specifically AWS CDK to be used for the task.

The project did not require new functionality other than IaC deployment system. The company had been satisfied with the performance and features of the old environments. All the manually configured features should still be maintained with the new system. A rough list of requirements was the following: use of IaC deployment system, version updates for application servers and database servers, preserving the current state of monitoring, preserving the current state of automatic scaling for the application servers. Listing of required version updates and features is presented in table 3. Out of these version updates, the Amazon Linux version update was the most important, as the old versions were close to a deadline where they would no longer receive automatic security updates. The other updates were necessary to ensure future compatibility with other services and applications.

Starting point	**Required state**
Amazon Linux AMI	Amazon Linux 2
Python 3.6,	Python 3.8
PostgreSQL 9.6,	PostgreSQL 13.3
Django 1.10.5	Django 3.2.9
Manually configured environment	IaC with AWS CDK

Table 3 *Necessary system updates.*

5.3 Implementation

5.3.1 Mapping of the existing infrastructure

Because of the lack of the up-to-date documentation, the process started with a careful mapping of the existing infrastructure. Each setting or a resource visible in the web console required work to make sure if it is actually in use.

5.3.2 Good practices

Design patterns were a source of confusion in the beginning. Simple questions like how to organize the infrastructure definition files into repositories and how to implement different definition files for testing, staging and production environments were puzzling. For the question about should the IaC definition files be kept separated from the other project files, it was found out that it is mostly a matter of taste. Some

teams use a single repository for all their projects. Implementing IaC definition files for different environments was first done with just copy-pasting a large part of the code to each files. It was quickly realized that this breaks the *don't repeat yourself* rule of programming. A better solution was later implemented with inheritance.

Separating testing to a completely different AWS account was an important practical realization. After being hesitant to make changes and spending time double checking relations between resources, this sped up the development process. Morris (2021) recommends running testing environments within the same account, but when learning to use the infrastructure deployment systems, separating the whole testing account can be the safest option.

Best way to run the deployment tools caused some considerations. On what degree should the tools be used by individual workstations? Individual workstations are only used by one person and tend to become the very definition of snowflake environments. Better solution would maybe be to use a virtual machine either locally within the workstation or as deployed in the cloud, but this does lead to accessibility problems. We ended up using local virtual machines to make sure the dependency on single workstation is avoided, but still doing the deployments by the workstations for convenience. Utilizing IaC tool like *Chef* (Chef Documentation, 2022) for individual workstation installations is interesting, since a small teams generally has only few workstations and losing one due to an accident is always a possibility.

5.3.3 Challenges

Much of the practical work had to be done because of the version updates of different modules. With this work, automated testing was invaluable and caught the majority of the bugs. A common source of the bugs was new and more strict requirements for some class definitions in some modules. Typically a new version of some class constructors required explicit definitions of some parameters. An interesting bug was noticed because of the Python version update. From Python version 3.8 a syntax warning is given when using "is" or "is not" comparison with integers or other literals. The comparison only works by coincidence in CPython implementations of the language for integers from -5 to 256, because of small integers and string caches.

One of the more difficult to solve bugs was caused by a specific module in the framework stack. This problem is mentioned in detail because it has to do with another declarative deployment system present in the project. The Django framework has a declarative system which deploys changes to the database structure. All the changes are made with migration files, which can be reverted if needed. This gives the system a transparent way to see all the changes done to the database. The migration files are compiled by the system by comparing the state of the database and the definition files called *models*. Any changes that the system finds are compiled

into migration files which can be then applied for the database. Developers of a certain module used in our application had made a mistake with the migration files. The migration files in the older version did not match the files, that were used in the latest version. Going back by reverting the migrations would have caused some data loss. Luckily the declarative system had features necessary for recovering such problem.

In a situation like this it does not matter how the database reaches the wanted state defined in the models, as long as it does so. Therefore it was possible to make new custom migration files and use the deployment systems ”faking” feature to fake that the migrations were made with the files included in the latest versions. Having the full history of all the database migration files might feel unnecessary for the developers. They might feel a temptation the squash some of them together and by doing so, make them more readable. Doing so might however leave somebody else depending on their code isolated and having no easy way to migrate their database state. Having migrations which make changes not only to the structure of the database tables, but also to the data, can make recovering from this kind of situation even more complicated.

5.3.4 Implementing the definition files

After finding out what kind of components the infrastructure was composed of, the first versions of the infrastructure stacks were written. The stacks were written as AWS CDK definition files written in Python. The language was chosen because it was the main programming language for our server side code. The use of a general purpose programming language with the definition files made it possible to use conditional logic. This was utilized to make creation of an object storage resource to depend on a parameter, to make it possible to use the same IaC definition files for creating a new environment from the scratch and also in a situation where old object storage would be used.

The project has a *path to production* implementation with separated environments for testing, staging and production. The usage of general purpose programming language made it possible to reuse parts of the code with inheritance. Inheritance is a concept in object-oriented programming where you can derive a child class from a parent class. The child class inherits all the features of the parent class, but can also implement new features or substitute some existing features. The main benefit of inheritance is avoiding repetition. Different environments for testing, staging and production were implemented as individual classes which inherited from a base class as shown class diagram presented in Figure 6. The base class can be seen mostly to hold references to representations of the cloud resources such as the database instance or the object storage bucket and the child classes contain

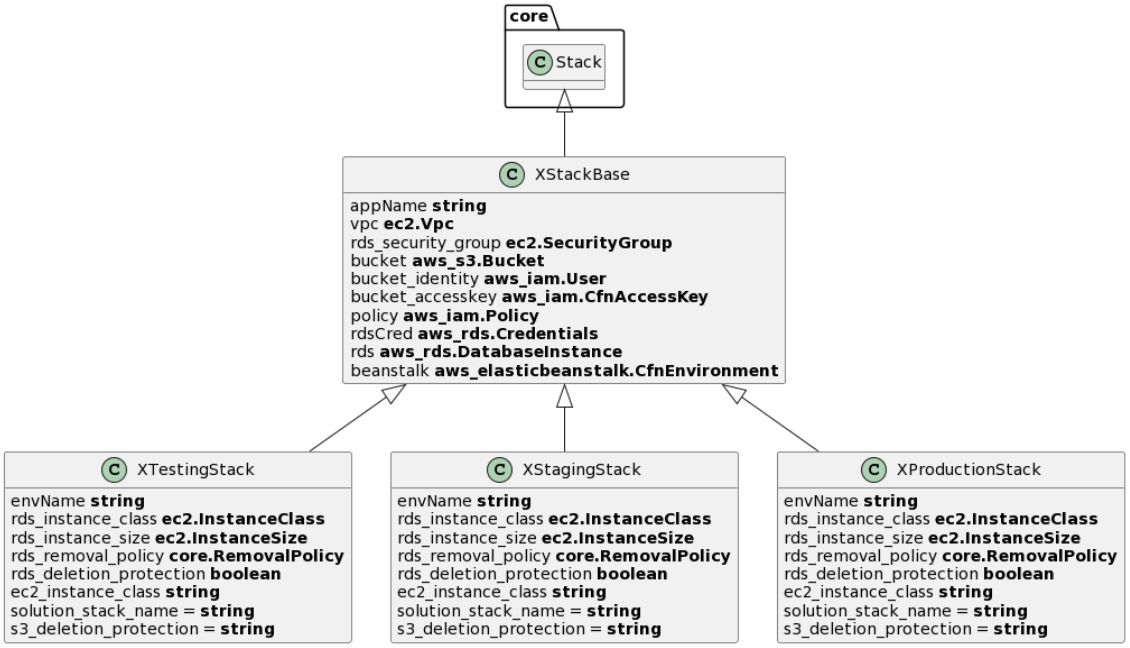

Figure 6 *Case study CDK Stacks class diagram.*

the fields for the settings necessary to make environments different in size where it matters for the costs. Code example 7 presents a stripped down version of the class definition used for the production stack. The contents of the base stack class are not included in this document for security reasons.

```
from stacks.x_stack_base import XStackBase
from aws_cdk import (
    core,
    aws_ec2 as ec2,
)

class XProductionStack(XStackBase):

    envName = "x-production"
    rds_instance_class = ec2.InstanceClass.STANDARD6_GRAVITON
    rds_instance_size = ec2.InstanceSize.LARGE
    rds_removal_policy = core.RemovalPolicy.RETAIN
    rds_deletion_protection = True
    ec2_instance_class = 'm6g.large'
    solution_stack = '64bit Amazon Linux 2 v3.3.9 running Python 3.8'
    s3_deletion_protection = True

    ...

    def __init__(self, scope: core.Construct, id: str,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
```

Code Example 7 *CDK stack written in Python following the structure of the production stack used in the case study.*

When all the problems and bugs were finally fixed and everything was working in the testing account, infrastructure deployments were done in the testing, staging and production environments. Some small configuration tasks were still left to be done manually. These included SSL certificate settings and DNS settings. Configuring the DNS settings was the last step as it was the way to point the existing customers clients to use the new environment.

5.4 Evaluation

The following subsections will cover each dimension of the S3MQual quality model introduced in the Chapter 4 from the point of view of the case study experience, while at the same time reflecting on literature especially in situations where the experience or sample size from the case study is not enough to justify a claim or when the case study came short of implementing some known good practice.

5.4.1 Accountability

Based on the experience of the case study, IaC has a positive effect on accountability. When the configuration files are kept under version control, it is possible to track precisely all the changes that were made to them and to see which developer made the change. It is also possible to see why some change was made by reading the commit messages. This is invaluable for debugging problems.

IaC seems to have a positive effect on the infrastructure documentation. The definition files themselves are an important part of the documentation, version control holds the historical data and explanations for changes and automated tools can be used to create graphs based on the definition files to support the documentation.

5.4.2 Availability

Based only on the case study, it is hard to say what kind of effect the IaC will have on the service's availability in the long run. The sample size is simply too small and the time period too short. In theory, it should be easier to maintain the state of the environment, and therefore the availability would not suffer much downtime, but there are some aspects that come into play when the IaC deployment systems are used by relatively small organization new to the methods.

In case study project, any major changes to the environment seem to happen quite rarely. This causes challenges to documenting the methods and processes of maintaining the system. If some urgent maintenance need would arise, there would be a temptation to use imperative commands with AWS CLI or by AWS Console to deal with the problem fast instead of making changes with the IaC definition files. This is not necessarily bad as Morris (2021) suggests that imperative methods are often used alongside the declarative systems. He does still recommend aiming for implementing all the infrastructure as code.

Writing the IaC definition files is arguably more difficult than clicking around in the WUI console. Therefore we can recognize a tendency for the developer to keep things more simple when implementing infrastructure deployments with IaC. Implementing some of the possible configurations to increase availability such as spreading the infrastructure across multiple physical locations, might get left out. This would hardly be the fault of IaC, but something that teams with smaller resources might have to deal with. Does implementing IaC force the team to settle for more simple infrastructure?

In general, it would still seem that the benefits of the transparency and the concept of Single Source of Truth for the environment will have a positive effect on availability in the long run.

5.4.3 Maintainability

Experience from the case study indicates that IaC has a positive effect on system maintainability although some of the positive effects appear only as new possibilities.

Case study project has a *path to production* implementation with separated environments for testing, staging and production. It is apparent that maintaining multiple environments is easier with IaC now when their definition files use common code. The amount of the environments is still quite small and the management they need is not very frequent, so the maximum benefit of the IaC implementation is not fully utilized. IaC makes the possibility to create more environments for data segregation more reasonable.

IaC does not necessarily make it directly easier to modify the environment. If some experimental settings have to be tried out it is probably still faster to use the web console or the AWS CLI. The benefits for to maintainability come from not having to deal with *snowflake* environments. Seems that using IaC deployment systems can cause some overhead for implementing modifications, but when they are implemented, the results are more robust.

Concept of modularity is related to maintainability, as implementation of replaceable modules makes it easier to apply changes to one of them while minimizing effects on the others. Morris (2021) stressed the importance of implementing IaC with modularity. Based on the experience from the case study it seems that it is not trivial to come up with such modules. It is better to split the definition files so that resources which might have different life cycle are created separately. This could mean separating database creation from application servers as their own infrastructure stack.

5.4.4 Scalability and Elasticity

Implementing scalability and elasticity becomes easier with IaC. In the most simple case the scalability can always be achieved by deploying multiple environments. That might not be the best solution for the problem, but with IaC that option always exists.

Morris (2021) presents two occasions when the same system is run in different environments. First one is the practice of implementing the *path to production*. In this approach organization implements isolated environments for testing, staging and production uses. Another use case is having multiple production environments for fault tolerance, scalability and segregation needs.

Scalability and elasticity in cloud can be implemented with setting up a system of load balancers and server clusters. In such system a load balancer works as a gateway node which distributes jobs for multiple application server nodes to handle.

The amount of the application servers can be increased dynamically to achieve scalability.

The case study used a PaaS product Elastic Beanstalk (Elastic Beanstalk documentation 2022) to set up such system. The configuration for the quantity of the application servers and the rules for the automatic scaling are implemented in the IaC definition files.

5.4.5 Performance

From the point of view of the team managing the environment, it is difficult to show a positive effect of IaC for performance. Any effect on performance would simply be a result from a better general understanding of the system and better choices made for the hardware configuration.

More obvious and direct effect on performance would probably come from implementing Continuous Delivery and Continuous Integration practices for the infrastructure code like suggested by Morris (2020), but those are outside the scope of this case study.

From end-customer point of view IaC might have an effect on performance as can adopting any software development good practices have. The more organized and effective the team producing the service is, the more performance the customer can expect.

5.4.6 Security and Privacy

Use of IaC practices should cause security and privacy considerations. Rahman and others (2019) recognized several common security issues that commonly occur in IaC scripts. They find out that one of the most common causes of security breaches is caused by storing secrets within source code. This was also pointed out by Morris (2021).

Not storing secrets within source code is especially important for future-proofing. Version control systems are meant to store the whole history of a project. For that reason it is not easy to remove information from the codebase once it has been introduced there. At least not without rewriting the history. Secret management systems should be used instead. They are part of storage resources that many cloud providers offer and cloud-agnostic options are available too (Morris 2021).

Another common security problem with IaC definition files, found by Rahman and others (2019), is the use of invalid IP address binding. When defining the networking configuration cloud developer might use the IP address 0.0.0.0 for the firewall or security-group settings. This address usually has a special purpose if used in such context. It means all possible addresses and can cause a security concern as

resources might allow connections from every possible network.

Use of HTTP without transport layer security, use of weak cryptography algorithms are more examples of common security problems in IaC definition files (Rahman et al. 2019).

The case study implementation was close to make two of the errors described above. The invalid IP address binding and the storing of secrets to the source code. This would have had an effect on the security of the database server. It would have been accepting connection attempts from any network for the specific port used by the database service. The database service would still have been protected by password, but the password could have been leaked in the future because it was stored to the IaC definition files. Even though the team is not planning to make the code repository public, the danger of having secrets within the source code must be recognized. Any local copies of the repository could leak the secrets.

It is possible to use static code analysis for the IaC definition files. Such software available for CloudFormation templates are for example CFripper and SonarQube. They promise to detect bugs and vulnerabilities, but analysis of their reliability is outside the scope of this study (CFRipper documentation 2022; SonarQube documentation 2022).

5.4.7 Usability

I'm going to evaluate the usability from the point of view of the team implementing IaC for their infrastructure. The usability from point of view of the customer would hardly have any change depending on what kind of infrastructure deployment system the service provider uses.

According to experience from the case study, the installability of IaC deployment tools is quite good in the sense that the methods and necessary steps are well documented. Installation process was not trivial as it required application-runtime installations, configuration files, and some familiarity with working with AWS services like AWS Identity and Access Management (IAM) is expected. The official installation documentation mentions this familiarity requirement, but the exact amount familiarity is difficult to estimate.

Operability of the commands is straightforward. There are only few commands for the basic use and once the user understand the basic idea of a declarative system, they are easy to use.

There is a somewhat steep learning curve when it comes to utilizing the constructs. The same seems to be true for the design patterns of the definition files, a topic which is not very well documented in the official AWS documentation, but given an emphasis in the book by Morris (2021). When working with the case study, it seemed smart to use also the management console GUI to inspect the changes

made to the system. Making testing with another AWS account and switching between them was easy once everything was initially configured.

5.4.8 Reliability

This dimension as many others above get benefit from the simple fact of the transparency of the state of the system. In case of a failure, there is one place to which go to check all the versions and all the components of the system. That alone makes fixing problems easier, because administrators can rely on the definition files correctly describing the environments. Consistency of multiple environments can be ensured as the different environments can share parts of the code.

We can also examine the reliability of the IaC deployment system itself. Are the deployments executed reliably, does the promise of idempotence hold true? Few times during the case study project, the deployments did not execute like they were expected. The result was an erroneous state, which had to be recovered from manually by deleting some resources or in some cases by redeploying the stacks.

First such situation occurred when a quota of public IP addresses was depleted. The problem was solved by removing test stacks reserving those addresses. This problem was simply resolved, but it did cause considerations for other factors that might cause infrastructure deployments to fail unexpectedly in the future.

Another situation when the case study experienced failed management commands was caused when executing a deletion command for a stack, which included resources with *deletion protection*. It is certainly expected that the system will then prevent deleting them, but it was interesting to notice that our IaC deployment system does not notice such thing in advance, but rather tries to delete everything else and only informs the user that certain resources will have to be deleted manually.

5.4.9 Features and Functionality

Defining infrastructure in the definition files does have a learning curve. On the other hand the maintenance of a more complicated infrastructure is made possible. Everything does not have to be implemented from the scratch. AWS offers a catalog of solutions which are delivered as IaC templates. The templates are available as CloudFormation templates and in some cases as AWS CDK code. Utilizing these solutions can have a significant positive effect on the features and functionality of a project.

To evaluate the features and functionality of the AWS CDK and CloudFormation themselves, more experience on competing technologies would be necessary. It would seem that the systems are quite mature, feature rich and extendable, but this is more of a intuitive feeling than an opinion based on facts.

Adopting IaC seems to have a positive impact on the motivation for our team to try out new things on the cloud. Even without initially implementing them with IaC. The fact that there is a way to *save* some infrastructure configuration as IaC definition files makes working with more complicated configurations less scary. Possibility to try out new things on a separate branch and to be able to come back to the original solution lessens the burden of manual documentation.

5.4.10 Recoverability

Earlier dimension about reliability addressed two cases where the management commands failed. One was caused by depleted public IP address quota and other was caused by configuration settings preventing deletion of a resource. In both situations the recovering from the problem was possible by doing the steps suggested or by inspecting error messages and acting based on them.

The case where *deletion protection* prevented complete deletion of an infrastructure stack, the process did take a long time (about an hour) to reach a state where the system was confident enough to inform about the erroneous state. For a deployment system which is running with such high privileges it is hard to justify why it should not be able to know such configuration. Therefore, it can be argued that the communication between the services could be improved and occasional failures with the IaC management commands should be expected.

Does adapting IaC practices have an effect on recoverability from the deployed applications point of view? To recover from a major problem like somebody accidentally or maliciously terminating all the servers, IaC could save the business by simply recreating the whole environment maybe even on another AWS account.

If everything was done perfectly following instructions by Morris (2021) implementing *all* the configurations as code, this would only take few minutes. The implementation of the case study did leave some aspects to be manually configured, but still the recreation the environments would only take a couple of hours instead of a couple of days it might have taken before.

5.4.11 Compliance

Using IaC practices does not prevent from making mistakes with security or implementing design antipatterns which make the project difficult to maintain. Using IaC in general seems to be becoming an *industry standard* practice though. IaC can help to adhering to regulations also. In case of external security auditions the company can easily present the structure of the networking of the infrastructure to prove that necessary steps are taken to prevent unwanted access to the system.

5.4.12 Empathy

Even though I have not experienced caring and attention from AWS personally when working with IaC systems, I am willing to recognize some empathy built into the system. This comes in the form of the wording of output messages and documentation. The impression could be because of a certain degree of assertiveness and seriousness in the warnings about possible security issues gives an impression that the system is built to care about the quality of the implementations and those concerns get interpreted as empathy.

To say that IaC would have an impact on our customer's impression of empathy seems far-fetched, but as the IaC practices make it easier to create segregated individual environments for different customers, such configurations could get interpreted as individual care and empathy.

5.5 Results

The analyse of the quality dimensions indicates IaC producing significant positive effects on service quality on almost all dimensions. Results are expressed in a compressed form in Table 4. The generalization of the results does depend on level of expertise of the team implementing and using the systems. Successful usage of the tools does not seem to require very advanced knowledge of some specific field, but rather the basic knowledge of many fields. These fields include: basic programming skills, working on commandline, details of different AWS services and understanding what they are used for, basic networking concepts, version control systems, keyfiles and certificates. Teams with a high level of expertise in the fields listed might not be affected by some of the found negative effects, while those with less experience might bump into them. The same is likely true for the positive effects. Teams struggling with the basic concepts might experience such friction that some of the positive effects are not easily achieved.

Positive effects	Negative effects
Accountability	
Changes can be tracked with version control.	
Actions can be traced with version control.	
Definition files themselves are documentation.	
Definition files can be used to generate graphical documentation.	
Availability	
Overall positive effects on quality increase availability.	Mistakes have a larger ”blast radius”.
Maintainability	
Maintenance of environments which use common components becomes easier.	Experimental settings might be faster to test with other tools.
Definition files form a single source of truth for the structure.	
Modular implementations are recommended and can increase maintainability.	
Scalability and Elasticity	
Always possible to implement by deploying multiple environments.	
Traditional ways of achieving scalability can be configured in IaC definition files.	
Performance	
Easier to make isolated production environments which do not interfere each other.	
Easier to update components and configurations that have an effect on performance.	
Security and Privacy	
Single source of truth to analyze security problems.	Definition files are a new attack vector
Manually configured environments can hide the security problems more easily.	
Static code analysis can find security problems in definition files.	
Usability	
High usability with multiple interfaces both graphical and commandline.	Initial configuration is not trivial.
Possibility to use general purpose programming languages.	Certain level of familiarity with related services is expected, but not well defined.
Good amount of documentation and literature available.	
Reliability	
Environments built with IaC are arguably more robust than manually created.	Deployment commands sometimes fail unexpectedly.
Features and Functionality	

Possibility to save configurations as IaC definition files encourage to experiment.	Testing out new configurations seems to be faster using imperative tools
Sharing and adapting solutions for infrastructure becomes significantly easier.	
Recoverability	
Possibility to recover by redeploying the infrastructure always exists.	Failing to implement modularity can cause problems if redeploy is necessary.
Compliance	
It is easier to produce documentation for security auditions.	
Easier to represent data about component versioning.	
Empathy	
Possibility to create segregated environments for individual customers.	

Table 4 *Combined results of the quality analysis.*

6 Conclusion

Working with deadlines and customers create time pressure for development teams. Choosing what to implement and when is under constant evaluation. Taking on a big development task can halt progress elsewhere. However investments on automation can make all the difference in long run.

Working with this project raised my interest for declarative deployment systems in general. The case study project used a database management system not greatly different from the IaC declarative deployment system. Exploring the similarities and the differences between the systems helped to understand the both. In both systems developers get to declare data structures as classes which in the end turn into either database tables or different infrastructure resources. When making updates and implementing changes, the deployment systems check for syntax errors or other kind of errors and then compares the current system state to the changes made. Both systems let the user to back away at this point, but this is also the point where the systems differ. The database management system creates a migration file which user can choose to apply or not. The IaC system probably uses similar files, but they are not visible for the user in the standard workflow. It would be interesting to map what exactly are the reasons this and the other differences between these systems.

Even though implementing IaC for the main parts of our server infrastructure was a successful endeavour, more could be done. New features that previously were difficult to implement are now within reach. The new approach did also change the way the team feels about the cloud infrastructure. Anything not defined with IaC feels much more temporary and fragile now. The bar for what is considered as well done is now higher.

In general IaC was evolved to solve problems with complicated manual infrastructure deployments. It brings software engineering methodologies and tools to cloud infrastructure deployments. It is a natural progression for the scripting practises that system administrators have used already for a long time, even though it does not render these tools unnecessary.

The key technologies behind IaC are virtualization, programmatic access to cloud services and for the most modern applications, the use of declarative deployment systems utilizing domain specific or general purpose programming languages.

Implementing IaC practises seems to have positive effects on service quality. It is quickly becoming an industry standard way of deploying resources in the cloud. Possibility to make repeatable well documented environments with declarative deployment systems is a powerful tool for modern system administrators. Especially larger teams benefit from increased accountability and version tracking of the environment.

Implementing IaC practises seems to be valuable even from the startup point of view. Fast prototyping typical for startup projects can benefit from IaC as ready made templates for different resource configurations are available as IaC definition files.

DevOps Engineer and technical consultant Esko Takku contributed to the work with the initial idea and a proof of concept version of the implementation.

References

Amazon Web Services documentation (2022). Available: `https://docs.aws.amazon.com/`. (Visited on 06/11/2022).

AWS CDK documentation (2022). Available: `https://aws.amazon.com/cdk/`. (Visited on 06/11/2022).

AWS CLI documentation (2022). Available: `https://awscli.amazonaws.com/`. (Visited on 06/11/2022).

AWS Lambda documentation (2022). Available: `https://aws.amazon.com/lambda/`. (Visited on 06/11/2022).

Azure Resource Manager documentation (2022). Available: `https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview`. (Visited on 06/11/2022).

Berry, Leonard L, A Parasuraman, and Valarie A Zeithaml (1988). "The service-quality puzzle". *Business horizons* 31.5, pp. 35–43.

Breitenbücher, Uwe, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger (Mar. 2014). "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA". In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, pp. 87–96.

CFRipper documentation (2022). Available: `https://cfripper.readthedocs.io/en/1.10.0/`. (Visited on 06/11/2022).

Chef documentation (2022). Available: `https://docs.chef.io/`. (Visited on 06/11/2022).

Craig, Iain D (2006). *Virtual Machines*. 1. Aufl. London: Springer Verlag London Limited.

Doolittle, Jeff and Robert Blumen (2022). "Luke Hoban on Infrastructure as Code". *IEEE software* 39.2, pp. 112–114.

Elastic Beanstalk documentation (2022). Available: `https://aws.amazon.com/elasticbeanstalk/`. (Visited on 06/11/2022).

Endres, Christian, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger (2017). "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), pp. 22–27.

Foote, Keith D. (Dec. 2021). "A Brief History of Cloud Computing". *Dataversity Digital LLC*.

Fowler, Martin (2010). *Domain-specific languages*. 1st edition. The Addison-Wesley signature series. Addison Wesley. ISBN: 1-282-79730-1.

Garg, Saurabh Kumar, Steve Versteeg, and Rajkumar Buyya (2013). "A framework for ranking of cloud computing services". *Future generation computer systems* 29.4, pp. 1012–1023.

Google Cloud Deployment Manager documentation (2022). Available: `https://cloud.google.com/deployment-manager/docs`. (Visited on 06/11/2022).

Google Cloud Platform documentation (2022). Available: `https://cloud.google.com/docs`. (Visited on 06/11/2022).

Hourani, Hussam, Mohammad Abdallah, and Abdelfatah A Tamimi (Nov. 2019). "A Proposed Cloud Computing Quality Model: S3MQual (Service Measurement Metrics Model)". *ICIC Express Letters* 13.11, pp. 1005–1014.

Hummer, Waldemar, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam (2013). "Testing Idempotence for Infrastructure as Code". In: *Middleware 2013*, pp. 368–388.

ISO-25010 (2011). *Software Engineering: Software Product Quality Requirements and Evaluation (SQuaRE) Quality Model and guide*. Standard. Geneva, CH: International Organization for Standardization.

JSON documentation (2022). Available: `https://www.json.org/`. (Visited on 06/11/2022).

Kumara, Indika, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel (2021). "The do's and don'ts of infrastructure code: A systematic gray literature review". *Information and Software Technology* 137, pp. 106593–.

Microsoft Azure documentation (2022). Available: `https://docs.microsoft.com/en-us/azure/`. (Visited on 06/11/2022).

Miguel, José P., David Mauricio, and Glen Rodríguez (2014). "A Review of Software Quality Models for the Evaluation of Software Products". *International Journal of Software Engineering and Applications (Chennai, India)* 5.6, pp. 31–53.

Morris, Kief (2021). *Infrastructure as code : dynamic systems for the cloud age*. 2nd. Sebastopol, California : O'Reilly Media, Incorporated.

Pulumi documentation (2022). Available: `https://www.pulumi.com/`. (Visited on 06/11/2022).

Rahman, Akond, Chris Parnin, and Laurie Williams (2019). "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, pp. 164–175. ISBN: 9781728108698.

Rahman, Akond and Laurie Williams (2019). "Source code properties of defective infrastructure as code scripts". *Information and Software Technology* 112, pp. 148–163.

Ruparelia, Nayan B. (2006). *Cloud Computing*. MIT Press.

Short, Jared (Aug. 2020). *CloudFormation, Terraform, or CDK? A guide to IaC on AWS*. Available: https://acloudguru.com/blog/engineering/cloudformation-terraform-or-cdk-guide-to-iac-on-aws. (Visited on 06/11/2022).

SonarQube documentation (2022). Available: https://docs.sonarqube.org/latest/. (Visited on 06/11/2022).

Synergy Research Group (2022). Available: https://www.srgresearch.com/articles/huge-cloud-market-is-still-growing-at-34-per-year-amazon-microsoft-and-google-now-account-for-65-of-all-cloud-revenues. (Visited on 06/11/2022).

Terraform documentation (2022). Available: https://www.terraform.io/. (Visited on 06/11/2022).

Vassallo, Daniel and Josh Pschorr (2020). *The Good Parts of AWS*.

XML documentation (2022). Available: https://www.xml.com/. (Visited on 06/11/2022).

YAML documentation (2022). Available: https://yaml.org/. (Visited on 06/11/2022).

Zheng, Xianrong, Patrick Martin, Kathryn Brohman, and Li Da Xu (2014). "CLOUDQUAL: A Quality Model for Cloud Services". *IEEE Transactions on Industrial Informatics* 10.2, pp. 1527–1536.