Tampere University

Antti Pessa

COMPARATIVE STUDY OF INFRASTRUCTURE AS CODE TOOLS FOR AMAZON WEB SERVICES

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
June 2023

ABSTRACT

Antti Pessa: Comparative study of Infrastructure as Code tools for Amazon Web Services
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Science
June 2023

Cloud computing has become an integral part of modern software development. Infrastructure as Code (IaC) is an approach to managing infrastructure through code instead of manual processes. This thesis presents a comparative study of two popular IaC tools, AWS Cloud Development Kit (AWS CDK) and Terraform, for managing cloud resources on Amazon Web Services (AWS). The study investigates the key features, functionality, and benefits of each tool, as well as their strengths and weaknesses for AWS development. The research methodology involved a literature review, a practical implementation with both tools and then a comparison with the use of software quality metrics. The main qualities compared were performance, maintainability, and developer experience.

The results show that both tools can define cloud infrastructure, have tools to support maintainability, and offer great developer experience. Terraform performed better in the performance comparison with faster infrastructure deployment and update operations. However, AWS CDK offers a higher level of abstraction, better integration with IDE tools, and allows developers to use their preferred programming language. The study concludes that AWS CDK is the preferred choice for IaC tool for AWS but recommends Terraform when working in multi-cloud environments or use cases where more mature tools are required.

Keywords: Infrastructure as Code, Cloud Computing, Terraform, Amazon Web Services, AWS Cloud Development Kit.

Contents

LIST OF ABBREVIATIONS

API	**Application Programming Interface**
AWS	**Amazon Web Services**
AWS CDK	**AWS Cloud Development Kit**
CDN	**Content Delivery Network**
CLI	**Command-line Interface**
DSL	**Domain-Specific Language**
DX	**Developer Experience**
FaaS	**Function as a Service**
GPL	**General-Purpose Language**
HTTP	**Hypertext Transfer Protocol**
IaaS	**Infrastructure as a Service**
IaC	**Infrastructure as Code**
IAM	**Identity and Access Management**
IDE	**Integrated Development Environment**
PaaS	**Platform as a Service**
REST	**Representational State Transfer**
SaaS	**Software as a Service**

1. Introduction

Use of cloud services is getting more and more popular each year. To manage all this, cloud infrastructure tools have been developed to ease deployment, increase security, and document the settings in the used infrastructure. When choosing what IaC tool to use in the project, what kind of factors should one consider? This thesis focuses on comparing two infrastructure as code (IaC) tools using Amazon Web Services (AWS).

Infrastructure as code refers to the practice of using code to manage and provision infrastructure resources, rather than manually configuring them through a web-based interface or command-line (Guerriero et al., 2019). This approach allows organizations to automate their infrastructure management processes, improve traceability and auditability, and reduce the potential for errors and downtime (Morris, 2021). For managing cloud resources on AWS, there are several IaC tools available, including Amazon's own AWS Cloud Development Kit (AWS CDK) and Terraform (AWS CDK documentation, 2023; Terraform documentation, 2023). While the tools accomplish the same tasks, they work in very different ways.

The goal of the thesis is to study and evaluate the effectiveness of IaC tools for automating cloud infrastructure. The objective of this thesis is to compare two different approaches to infrastructure as code for AWS: AWS CDK and Terraform, to determine which is better suited for different use cases. Research aims to answer the following questions:

- **RQ1: What software qualities are important for an IaC tool?**
- **RQ2: What are the key features and functionalities for IaC tools using AWS?**
- **RQ3: How does AWS CDK compare to Terraform?**

This thesis aims to find answers to the research questions presented through the use of a comparative study. Comparative studies are investigations to analyze and evaluate, with quantitative and qualitative methods, a phenomenon, subjects or objects to detect similarities or differences (Coccia & Benati, 2018). The comparative study is performed using the case study method. Case study allows for learning concrete, contextual, in-depth knowledge about a specific real-world subject (McCombes, 2019). In this thesis, we will

compare and evaluate these two tools in terms of their features, capabilities, and performance, with the aim of providing insights and recommendations for organizations and developers looking to adopt IaC in their AWS environment. Overall, we want to help developers make informed decisions about which tool is best suited for their needs.

The remainder of the thesis is structured as follows. Chapter 2 covers the history of cloud computing, the different cloud service models and the three main cloud providers currently: Amazon, Microsoft, and Google. Chapter 3 focuses on infrastructure as code: how it has evolved, different programming paradigms it encompasses, quality characteristics and practices for adoption. Chapter 4 delves into the two tools being compared AWS CDK and Terraform. It goes through the tools key features and functionalities. Chapter 5 explains the case study designed for comparing AWS CDK and Terraform. In Chapter 6 comparisons are made between the two tools regarding performance, developer experience and maintainability. Chapter 7 ends the thesis with the conclusion.

2. Cloud Computing

The emergence of cloud computing represents a fundamental change in the way IT solutions are invented, developed, deployed, scaled, updated, maintained and paid for (Marston et al., 2011). In this chapter we look at how cloud computing got started, where it is now, what different types of cloud computing there are and then look at the three biggest cloud providers.

2.1. Brief history of cloud computing

Before cloud services existed, hosting a website on the internet required setting up physical hardware and servers. This was a tedious task and one that required a lot of upfront cost and technical knowhow. As more and more businesses needed websites it opened a business avenue for companies to offer hosting and maintenance services.

In the word “Cloud Computing”, “Cloud” means a carrier or provider who provides the services over the Internet. “Computing” is the processing, computations, calculations, or various resources that are provided by a computer (Surbiryala & Rong, 2019). Rajaraman (2014) prefers the term computing utility, which describes the new model of “pay for what you use” computing. This was foreseen by John McCarthy at MIT in 1961 when he said: “If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility.” (Rajaraman, 2014; Surbiryala & Rong, 2019).

Cloud computing has its origins in the 1960s with the invention of time-sharing systems which offered programmers a shared resource. Before programmers typed the code on punch-cards and then submitted it to the computer which executed it synchronously. This lead to a lot of waste resources, because the computer was idle for most of the time (Krishnan, 2010). The idea was to create a network of computers that could work together to solve complex problems, with each computer contributing its processing power to the task. This concept is known as distributed computing.

In the 1990s, the Internet became widely available and the first web-based services, such as online storage and web-based email, were introduced. This paved the way for the development of cloud computing, which allowed users to access these services from any device with an internet connection. One of the first companies in this field was Salesforce with their SaaS product that provided customer relationship management for their users.

In the 2000s came the first IaaS offerings from AWS and PaaS services such as Google's Application Engine (Surbiryala & Rong, 2019).

Today, cloud computing is a cornerstone of modern business and is used by organizations of all sizes to access a wide range of computing resources and services, including storage, computing power, and software. The cloud has become an essential tool for businesses to increase efficiency, reduce costs, and remain competitive in a rapidly changing technological landscape.

2.2. Cloud service models

2.2.1. Infrastructure as a Service

Infrastructure as a service (IaaS) is the base layer and foundation of the cloud computing service model, offering a computing platform with virtual server space, bandwidth, network connections, IP addresses, and load balancers (Freet et al., 2015; Kavis, 2014; Mustafa & Zeebaree, 2021). With IaaS, tasks related to managing a physical data center and infrastructure servers, disk storage, networking, are abstracted and offered as a set of services that can be managed and automated through code- or web-based consoles. While developers are responsible for designing and coding applications and administrators must install, manage, and update third-party solutions, the physical infrastructure is no longer a concern. (Kavis, 2014)

Figure 1 displays how each cloud service model provides an extending level of abstraction and automation allowing the customer to focus on the business problems they want to solve. IaaS provides virtualized computing resources, allowing users to manage and control the underlying infrastructure. PaaS abstracts away the infrastructure, providing a complete platform for application development and deployment. SaaS offers fully managed software applications, eliminating the need for users to manage infrastructure or platform components.

The most used and with the most expansive offering of IaaS services out of all cloud providers is AWS (Kavis, 2014). Their Elastic cloud compute (EC2) service delivers scalable computing capacity with customer specified operating system (OS), memory and storage (AWS EC2, 2023; Mustafa & Zeebaree, 2021).

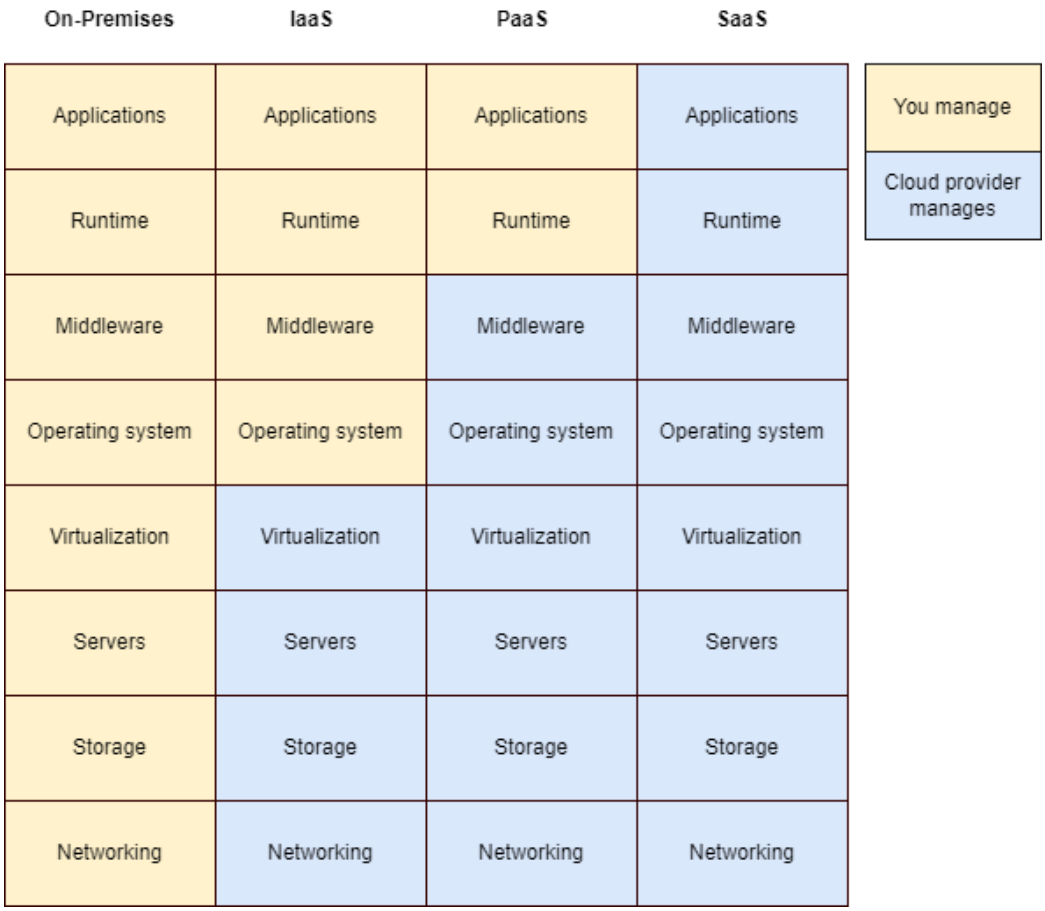

Figure 1. Cloud service models.

2.2.2. Platform as a Service

The National Institute of Standards and Technology (NIST) defines Platform as a Service (PaaS) as:

> "The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment."

PaaS is the next level of abstraction from IaaS for the application stack. It provides developers with access to different software building blocks to create new applications (Freet at al., 2015). For instance, developers creating highly scalable systems often must

write extensive code to manage caching, asynchronous messaging and database scaling, many PaaS solutions offer those capabilities as a service which frees developers to focus on solving the core business problems (Kavis, 2014). Freet at al. (2015) state the main benefits of PaaS over traditional application development and deployment models are server and storage overhead, network bandwidth, software maintenance, support personnel and lower skill requirements.

Kavis (2014) notes one of the downsides of PaaS that the developers have minimal control of the underlying software controls like memory allocation and amount of cache or number of threads. PaaS vendors can even throttle how much a service consumer can use so that the vendor can ensure the platform scales equally to everyone. PaaS pioneers like Google App Engine and Microsoft Azure dictated the platform stack and the underlying infrastructure for developers. Originally Google Apps Engine demanded that developers wrote their code using Python and host it on Google data centers, whereas Azure originally mandated the use of .NET technologies on Microsoft data centers. Nowadays both vendors support multiple programming languages. (Kavis, 2014)

2.2.3. Software as a Service

Software as a service (SaaS) is a service delivery model that delivers complete applications to the end user over the Internet (Freet et al., 2015; Kavis, 2014). As seen from Figure 1, only responsibility left for the customer is user management, everything else is left to the SaaS vendor. This also means the customer has no control over the underlying infrastructure, network, operating system, or even individual application capabilities. With the SaaS model, the cloud provider makes an instance of the application that the customer can connect to and use through a browser (Freet et al., 2015).

Freet et al. (2015) describes the key characteristics of SaaS as multi-tenant architecture, easy customization, and improved access. SaaS models advantages include easier administration, patch management, easier collaboration, compatibility, and global accessibility. Common SaaS applications include customer relationship management (CRM), enterprise resource planning (ERP), payroll, accounting, and other business software (Kavis, 2014).

2.3. Cloud providers

Central to the cloud computing paradigm are cloud service providers, which offer a wide range of services and resources to enable the deployment and management of applications and infrastructure in the cloud. This chapter introduces the three major cloud providers: Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

2.3.1. Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform that provides a wide range of services, including computing power, storage, and databases. AWS started in 2006 with offering their IT infrastructure services to businesses in form of web services, later to be known as cloud computing. Nowadays AWS is powering business in 190 different countries and has data centers located all around the world. (AWS, 2023)

In a 2022 report by Gartner (2022) AWS was named the leader of the magic quadrant for cloud infrastructure and platform services (CIPS). The scope of the magic quadrant for CIPS includes infrastructure as a service (IaaS) and integrated platform as a service (PaaS) offering. Quadrant leaders are providers who distinguish themselves from the competition by offering a service suitable for strategic adoption and can serve a broad range of use cases. The quadrant can be seen in Figure 2 with AWS holding the top spot followed by Microsoft and Google. The report notes future focus for AWS expanding to new territory with partnerships with telecoms. (Gartner, 2022)

Gartner (2022) note the breadth of functionality as a strength for AWS. They are seen a standard setter, developing, and establishing methodologies that are copied by the competition. They are also the market-share leaders with double the revenue compared to their closest competitor Microsoft (Gartner, 2022). According to data from Synergy Research Group (2022) AWS holds 34% of the Q3 2022 cloud infrastructure service market share. Microsoft and Google held 21 % and 11 % respectively, so in total 66% of the market belongs to these 3 cloud providers. Another strength is the vibrant and prosperous ecosystem around AWS. Independent software vendor (ISV) partners prioritize supporting AWS first due to their market share and momentum. Such partners include SAP, Splunk and VMware. (Gartner, 2022)

Even though AWS holds the top spot they are not perfect. Gartner (2022) see AWS optimizing for short term gain when dealing with customers, which is leading to eroding

customer relationships. Another weakness is AWS lack support for multi-cloud and sovereign solutions. AWS had a prolonged outage on December 7th 2021 which impacted customers in various regions across the world. It revealed some multi-region dependencies on the internal AWS network, which is hosted on the us-east-1 region. That region also hosted support ticketing for North America, which resulted in customers having difficulty communicating with AWS support during the outage. These problems along with AWS failure to communicate properly about the outage, left a lot to improve for AWS. (Gartner, 2022)

Figure 2. Magic Quadrant for Cloud Infrastructure and Platform Services (Gartner, 2022).

2.3.2. Microsoft Azure

Microsoft announced Azure, then called Windows Azure at its Professional Developers Conference in October 2008. The first service hosted on Azure was SQL Azure relational database was announced in 2009, afterwards came support for PaaS websites and IaaS virtual machines in 2012. With the vision of Microsoft expanding its proprietary borders from the new CEO Satya Nadella, Windows Azure was renamed to Microsoft Azure in 2014. He wanted Microsoft to embrace open-source technologies such as Linux, which they had formerly considered as hostile competitors. (Warner, 2020)

In the Gartner 2022 Magic quadrant report Azure was strong in all use cases, including cloud and edge computing. Azure is seen as a good fit for organizations already using other Microsoft products such as Office365. Microsoft is also praised for its investment in hybrid and multi-cloud, along with making architectural and security improvements to their cloud platform. Most of Azure's clients tend to be midsize and large enterprises. Azure keeps closing the market share gap on AWS, which has already shrunk significantly in Europe. Microsoft Azure strategy is solution oriented with broad range of Microsoft cloud capabilities and ecosystem partners to satisfy customer needs. (Gartner, 2022)

Weaknesses in Azure found by Gartner (2022) include lack of novel innovations in the market for cloud infrastructure and platform services. Azure customers also faced continuous stream of reliability and security related incidents. Non-transparent pricing is also a talking point. Gartner clients report their Azure costs increasing over time without knowing the reasoning. Critical features are often only included in the premium tier and Azure's native cost management capabilities lag behind competitors in maturity and completeness. (Gartner, 2022)

2.3.3. Google Cloud Platform

While AWS launched in 2006 with IaaS offerings, Elastic Compute Cloud virtual machines and simple storage service (S3) elastic object storage, Google launched their cloud ventures in 2008 with their PaaS offering App Engine (AWS S3, 2023; Google App Engine, 2023). App Engine is a fully managed, serverless platform for developing and hosting web applications. In the early 2010s large organizations were hesitant to move to the cloud and saw the IaaS route as the path of least resistance. (Srinivasan, 2018)

Gartner's 2022 report marks Google as a magic quadrant leader with strong showing in different categories and they've made significant improvement in their edge capabilities. Google has invested in being a broad-based provider of IaaS and PaaS by expanding its capabilities. Google operations are geographically diversified, and its clients vary from startups to enterprises. Strengths for Google include highest revenue and capability gains of any provider in the CIPS market. This is the result of increase field sales, co-selling with partners and commitment to offering a competitive platform in terms of capabilities. Another strength is Google's flexibility with sovereign cloud partnerships outside its core regions. (Gartner, 2022)

Gartner warns about Google clouds increasing prices as their prices might not stay low forever. Google has historically attracted clients through its aggressive pricing compared to the competition. First signs of this were when Google increased prices by 100% of its storage services. They are also causing confusion among customers with their two customer enablement programs Rapid Assessment & Migration Program (RAMP) and Cloud App Modernization Program (CAMP). Even though its revenue gains GCP is the only market share leader that continues to operate at a large financial loss. Google keeps investing heavily into the cloud, but there have been other examples where Google has given up on products such as Google+ and Google Glass. (Gartner, 2022)

3. Infrastructure as Code

This chapter discusses the concept of infrastructure as code, different approaches: declarative and imperative, requirements and challenges in adopting IaC tools, and software quality attributes to measure them.

3.1. The need for infrastructure as code tools

One of the reasons for IaC came from the problem caused by the ease of provisioning new cloud resources. In many companies it created ever-growing catalog of systems which required more and more time to maintain (Morris, 2021). Without infrastructure as code tools, it is very time consuming to inspect the current configuration of the system and making changes to it becomes difficult. Documentation of changes and reasons behind changes are often hidden when cloud configuration changes were made manually through interactive configuration tools. Manual configuration of infrastructure can lead to misconfigurations. Manual changes increase the change failure rate of systems, slow development and expose the system to potential security exploits (Wang, 2022). Due to these problems a solution had to be developed.

Guerriero et al. (2019) define infrastructure as code as DevOps tactic for managing and provisioning infrastructure through machine readable definition files, rather than physical hardware configuration. IaC is a practice to infrastructure automation based on practices from software engineering, focusing on consistency, repeatable routines for provisioning and changing systems and their configuration (Morris, 2021). Wang (2022) describes IaC as applying DevOps practices to automating infrastructure changes in a codified manner to achieve scalability, resiliency, and security. Figure 3 presents how IaC applies the CAMS (Culture, Automation, Measurement, Sharing) DevOps model as part of automating infrastructure through: code as documentation, version control, software patterns and continuous integration. DevOps is a culture shift towards collaboration between development, quality assurance, and operations. It integrates the two worlds using automated development, deployment and infrastructure monitoring. (Ebert et al., 2016)

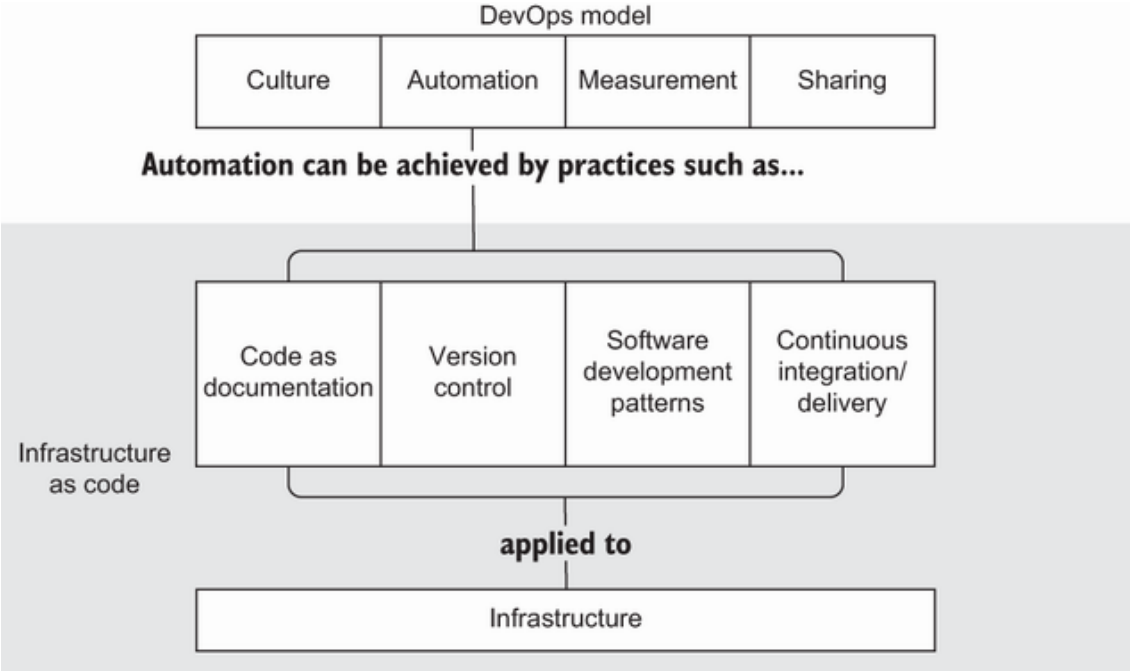

Figure 3. IaC a DevOps tactic for Infrastructure (Wang, 2022).

Wang (2022) lists four of the most important principles for IaC: reproducibility, idempotency, composability, and evolvability. Reproducibility ensures we can use the same configuration to reproduce an environment or infrastructure resource. The principle of idempotency ensures you can repeatedly run the automation on infrastructure without affecting its end state or having any side effects. Idempotency is important to safeguard against creating too many resources or destroying critical infrastructure, such as a database. Composability ensures that we can assemble any combination of infrastructure resources and update each one without affecting the others. The evolvability principle ensures that you can change infrastructure resources to accommodate system scale while minimizing effort and risk of failure. (Wang, 2022)

Over time many different languages have been developed each dealing with a specific aspect of infrastructure management. From tools able to provision and orchestrate virtual machines such as Cloudify and Terraform, to those doing a similar job with respect to container technologies among them Docker Swarm and Kubernetes, to machine image management tools like Packer, to configuration management tools such as Chef, Ansible and Puppet (Ansible, 2023; Chef, 2023; Cloudify, 2023; Docker, 2023; Kubernetes, 2023; Packer, 2023; Puppet, 2023; Guerriero et al., 2019). This thesis focuses on two tools for managing cloud infrastructure, AWS CDK and Terraform.

3.2. Different approaches to IaC

There are two main programming paradigms to IaC languages are declarative and imperative, also called procedural (Kumara et al., 2021). Declarative languages try to describe the result rather than the steps required to achieve it. With IaC languages this means describing the desired end state of the environment. Declarative IaC languages include Terraform and CloudFormation. (AWS CloudFormation, 2023; Birkman, 2022; Kumara et al., 2021)

Imperative, also called procedural IaC languages specify the steps required to achieve the desired state of the infrastructure resources. In imperative languages, the user specifies the sequence of actions that must be taken to reach the desired state of the infrastructure. Imperative IaC languages include tools such as AWS CDK, Pulumi, Puppet, Ansible and Chef. (Birkman, 2022; Kumara et al., 2021; Pulumi, 2023)

Both paradigms have their advantages and disadvantages. Birkman (2022) notes with the use of declarative approach in Terraform the code always represents the latest state of the infrastructure. While with Ansible templates, the scripts do not represent the current state of the infrastructure, the developers need to know the full of history of every change that has ever happened. This also makes it easier to make reusable code, allowing developers to focus on describing the current state of the infrastructure.

A new trend with IaC is the use of general-purpose programming languages (GPL) to manage infrastructure. Examples of this are Pulumi and AWS CDK, which both support languages such as Java, JavaScript and Go. The other approach is through domain-specific languages (DSL). These include Terraform, with HCL, CloudFormation and Openstack use YAML or JSON. GPLs are meant for a specific domain, whereas GPLs can be used in variety of domains. DSLs are easier to learn, since they deal with just one domain compared to GPLs. Birkman (2022) also notes that DSLs are clear and more concise as the languages are built to do one thing. DSL code usually uses a uniform, predictable structure, which make it easier to navigate and understand. GPLs have the advantage of not having to learn a new language. with For example if a developer has a JavaScript background it is faster to learn Pulumi than learning Terraform with HCL. GPLs also have bigger ecosystems and more mature tooling than typical DSL. This effects the number and quality of IDEs, libraries, testing tools. They also allow for more power and functionality than DSLs through control logic, automated testing, code reuse, abstractions, and integration with other tools. (Birkman, 2022)

3.3. IaC quality characteristics and best practices

In their conference paper “Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry” Guerriero et al. (2019) conducted 44 semi structured interviews with senior developers. Their aim was to find out about the developer perspective on pros and cons on current IaC tools and challenges faced while developing infrastructure code. From the interviews they could gather four best practices.

- Secret injection: Parameterize everything, avoid hardcoding so the user or orchestrator can inject their desired input.
- Break-fast: Program the infrastructure to be buildable and breakable as fast as possible. This is to combat one of the disadvantages of developing IaC, the long feedback loop between changes to scripts and deployment of the infrastructure.
- Reuse by abstraction: Making scripts recall each-other to allow for interdependency and interchangeability.
- Low-nesting: Limit tree of nodes generated by scripts to maximum of one recall.

Most of these practices are inherited from standard software development good practices while considering the declarative nature and tree-like structure of many IaC tools (Guerriero et al., 2019). Journal article “The do’s and don’ts of infrastructure code: A systematic gray literature review” by Kumara et al. (2021) echo the above sentiments. They mention also focusing on writing IaC code that is readable by humans, not computers. This is accomplished through use of consistent naming, formatting and making parameters and their types explicit. Another practice mentioned by Kumara et al. (2021) is documenting little but well. The source code should work as documentation, it ensures that when the code is stored in a central repository it is always up to date. Use of document templates is also recommended as it produces more consistent documents.

In terms of IaC tools used by the developers surveyed the most popular tool was Docker with 59.0% usage among the developers. Most popular cloud based IaC tools were Ansible, Chef and Terraform. The survey by Guerriero et al. was conducted in 2019 which explains why newer tools such as AWS CDK and Pulumi were not mentioned.

Guerriero et al. (2019) used six categories to rate the pros and cons of these different IaC tools:

- Coding: How easy it is to build scripts with it being declarative or imperative.
- Portability: How easy the scripts are to port to different target infrastructure, for example from configuring Linux nodes to Windows nodes.
- Automation: What level of automation the tool gives in building infrastructure.
- Usability: How easy the tool is to use; does it provide a local development or testing environment or use higher abstractions to designing infrastructure.
- Extensibility: How well the tool can be used outside of its original context and does it provide ability enhance it through other platforms.
- Maturity: How stable the software is, how wide the adaption rate is and how often it gets updated.

The developer comments on each category were split into positive and negative which were both counted and the total ratio was divided *(ratiotool = (pos − neg)/total)* to get the overall feeling around a specific tool (Guerriero et al., 2019). Most popular tools were Powershell, Packer and Saltstack which however had only few mentions which indicates low popularity. Negatively rated tools were Puppet and Chef but garnered more comments both negative and positive which indicated higher usage and adoption rate. Guerriero et al. (2019) summarize their findings that there not being a perfect tool for IaC as all tools have their own cons and it comes down to experience in canceling out the cons and getting most out of the pros. I agree with the sentiment of no perfect solution and see it being more of a question of preference while still choosing tools which are widely enough adopted and are getting regular updates. In the cloud ecosystem services are ever changing and new services are launched, so it requires a tool which is keeping up with them.

3.4. Requirements and challenges for adopting IaC

Through interviews Guerriero et al. (2019) gathered a list of most common issues found in IaC development:

- Testability: Testing IaC code is nearly impossible and code-review guidelines are lacking.
- Readability/Polyglot: Multitude of formats and specialist knowledge needed when working with IaC.
- Inconsistency: Newer versions of tools might break backwards compatibility and might not be supported by other used tools.
- Runtime automation: Runtime automation is still limited to just changing elasticity and scalability properties.
- Portability: Difficult to port existing scripts between technologies.
- Concurrency: Race-conditions and circular dependencies in CI pipelines with testability also being poor.
- Lack of IDE: Missing a development environment with proper tooling for IaC.

Out of the gathered issues testability was in the forefront of many developers minds. One developer noted: “Issues are mostly related to setting up a testing environment, since this is usually quite a complex problem. Combine that with no standard practices when it comes to testing and you have one big mess” (Guerriero et al., 2019). A testing framework for IaC along with static analysis tools and language standardization were most wished for features to the IaC ecosystem.

Challenge facing IaC developers is the non-trivial relationship between readability and polyglotism in IaC blueprints and their portability. There is a tradeoff between these two challenges which not fully understood and requires further study (Guerriero et al., 2019). Polyglotism refers to the practice of writing code in multiple languages to gain additional functionality and efficiency not available in a single language (Techtarget, 2023). The use of multiple languages within IaC is a factor to be cautious about, yet it is also an unavoidable reality. For example, Docker, Terraform and GitHub Actions perform three completely different roles regarding infrastructure and warrant their own languages. Docker allows for packaging software through OS-level virtualization, while Terraform

focuses on managing cloud infrastructure and GitHub Actions (2023) is a continuous integration platform. Focusing on code portability can result in a more limited set of tools and technologies being used, which can lead to code that is less expressive and more difficult to understand. Ultimately, organizations must strike a balance between polyglotism, readability, and portability in their IaC tools, considering the specific needs and goals of their operations. The goal should be to write code that is both maintainable and understandable, while still being able to be used in a variety of environments.

3.5. Measuring IaC tools through software quality metrics

While the comparative study will have subjective opinions and personal bias can play a role in the comparison, having metrics and data to compare is important for the study. With the ISO/IEC 25010 quality model, different quality characteristics can be studied, and metrics can then be used to measure those characteristics between the two tools. The model determines what characteristics should considered when evaluating the properties of a software product. It includes eight quality characteristics, which are shown below in Figure 4.

Figure 4. Software product quality model (ISO, 2023).

Performance is one of the main areas to compare between IaC tools. This includes measuring the time it takes for each tool to provision resources and execute infrastructure changes. ISO/IEC 25010 quality model (2023) splits performance efficiency into three characteristics: time behavior, resource utilization and capacity. Time behavior is the degree which the response and processing times it takes when performing functions. Resource utilization describes the number and types of resources used by a product. Capacity is the degree which maximum limits of a product meet requirements. (ISO, 2023)

Portability is described by the ISO/IEC 25010 as the degree of effectiveness and efficiency with which a product can be transferred from one hardware or other operational or usage environment to another. Sub-characteristics include adaptability, installability and replaceability. Adaptability is how well a product can effectively be adapted for different or evolving hardware, software, or other environments. ISO (2023) notes adaptability includes scalability of internal capacity of the product. Scalability of a IaC tool is important to compare to see how well it can handle large scale infrastructures. Deployment time is an important metric for evaluating scalability, as when traffic increases and more resources are required the deployment time can be a bottleneck that limits the scalability of the system.

Maintainability represents the degree of effectiveness and efficiency with which a product can be modified to improve it, correct it, or adapt to changes in environment and in requirements (ISO, 2023). Testability is a sub-characteristic of maintainability in the ISO/IEC 25010 quality standard. Software testability is the degree to which a software system or a unit under test supports its own testing (Garousi et al., 2019). Garousi et al. (2019) have classified the definition software testability to three groups: ease of testing, ease of revealing faults and other focus areas. They conducted a survey in the form of a systematic literature mapping to find out how to measure and improve testability. IaC tools can be measured in testability by the support for testing frameworks, the ability to write and run tests easily, and the availability of mocking and simulation tools. Both AWS CDK and Terraform support testing frameworks and the ability to write and run tests. However, the testing frameworks and tools available differ between the two tools.

Software complexity is another important factor to measure through metrics. Personal bias can have an effect in the comparison, so having metrics will give a more fair end result. Dalla Palma et al. (2020) have cataloged software quality metrics to evaluate infrastructure code. The study purposed 46 metrics to identify IaC properties focusing on Ansible, which is another popular IaC tool. While the study focuses on finding quality metrics for Ansible, it also found metrics for generic evaluation of IaC tools. The metrics were split into three categories: language-agnostic code characteristics, metrics previously developed for Puppet and metrics concerning Ansible. Language-agnostic metrics include LINESBLANK, LINESCOMMENT, LINESSOURCECODE that measure complexity of a source file. Higher number of lines can affect the maintainability of the code. NUMCONDITIONS is the count of Boolean expressions without Boolean operators and

NUMDECISIONS is the count of Boolean expressions composed of conditions and one or more Boolean operators. (Dalla Palma et al., 2020)

ISO/IEC 25010 model (2023) specifies usability as: “Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.”. Usability is composed of many sub-characteristics, of which learnability could be the focus in this thesis. In a survey on software learnability Grossman et al. (2009) studied different definitions for learnability. Learnability is hard to define, and it has many different definitions and meanings. Two main definitions found are initial learning of a system or software and extended learning, which includes the initial learning and the long-term learning for mastering the use of the software. Measuring initial learnability can be done through picking users who are not used to a system and measuring the time it takes them to reach a specified proficiency in using it. Studying extended learnability can be harder as the testing would take place over a longer period of time, potentially multiple years. Generic usability is linked with learnability as how easy to use a software is. Ease of use could be compared through the clarity and readability of the configuration language, as well as the availability of documentation, tutorials, and community support. (Grossman et al., 2009)

Not part of the ISO/IEC 25010 quality model, but related to usability is developer experience. Fagerholm & Münch (2012) defined the term Developer Experience (DX) as a means for capturing how developers think and feel about their activities within their working environments. DX consist of experiences that developers encounter during their involvement in software development, including a broad range of artifacts and activities. These experiences can be split into three categories: development infrastructure, feelings about work and the value of one’s own contribution. Development infrastructure consists of experiences with development and management tools, programming languages, libraries, platforms, frameworks, processes and methods. (Fagerholm & Münch, 2012)

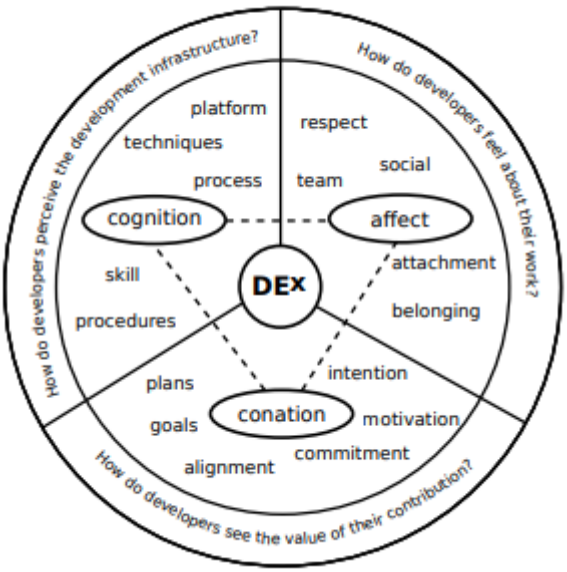

Figure 5. Developer Experience: Conceptual framework (Fagerholm & Münch, 2012).

Fagerholm & Münch (2012) have designed the conceptual framework shown in Figure 5, which describes DX as an interaction between cognitive, affective, and conative factors. The cognitive factors include the concrete interactions with development tools and the execution of the software process. Perceiving them in a positive light will likely contribute to a better DX. The affective factors influence how developers feel about their work, while the conative factors affect how developers see the value of their contribution.

AWS advertised AWS CDK for the improvement in DX due to allowing developers use modern programming languages to define their AWS infrastructure in a predictable and efficient manner (Fife, 2018). DX shares a lot of metrics with usability, but some metrics to focus with DX are learnability, community support, tooling, productivity, adoption rate and flexibility. Adoption and community support go hand in hand, as the more users there are for the product, the bigger the community gets. Tooling can be compared by evaluating the IDE support available for each IaC tool.

4. Cloud IaC Tools

This chapter will discuss the cloud infrastructure as code tools compared in this thesis along with AWS CloudFormation, as it is used by AWS CDK. For each tool key features and functionalities are examined.

4.1. AWS CloudFormation

AWS launched CloudFormation in 2011 as a way for developers to create and provision collections of related AWS resources in an orderly and timely fashion. AWS wanted to reduce time spent on managing infrastructure resources and increase time working on the applications. User creates a template that describes all the AWS resources they want and then CloudFormation takes care of provisioning and configuring those resources. Many of the third-party declarative IaC tools that exists for AWS leverage CloudFormation under the hood, offering extensions or augmenting functionality provided by CloudFormation. (AWS CloudFormation, 2023; Campbell, 2020)

CloudFormation simplifies infrastructure management through having all your resources in one single unit, everything gets created and deleted at the same time. Configuring the same resources through the AWS web console can be a complex, long and tedious process. Another feature with CloudFormation is the ability to replicate your infrastructure to multiple regions, again saving time from having to do manual configurations. It also allows users to easily control and track changes in the infrastructure. With the templates being text files, they can be stored in version control and know exactly what changes have been made and reversing back to previous versions is easy. (AWS CloudFormation, 2023)

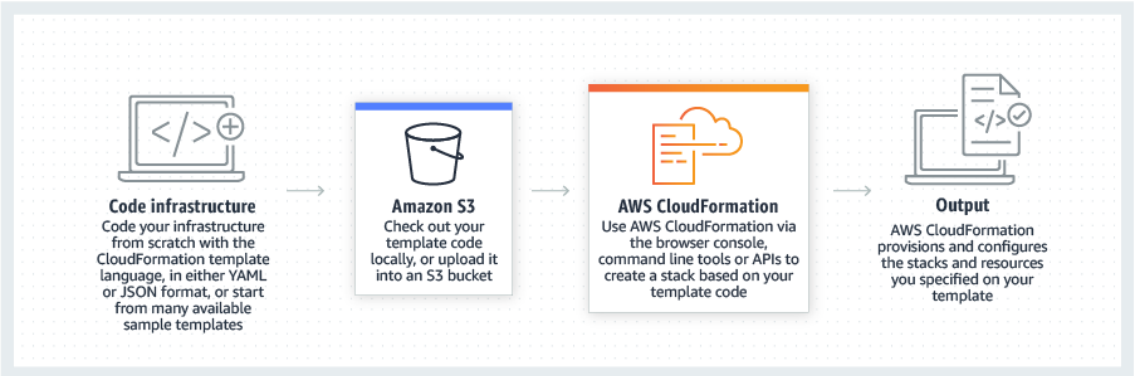

Figure 6. CloudFormation workflow (AWS CloudFormation, 2023).

CloudFormation uses JSON or YAML templates to model the desired infrastructure. These templates can be stored in an S3 bucket or passed directly to the CloudFormation service. Figure 6 represents the CloudFormation workflow. After CloudFormation receives the template it creates a stack, which consists of all the infrastructure specified in the template. When updating a stack CloudFormation generates a change set, which is a summary of the purposed changes. (AWS CloudFormation, 2023)

```yaml
AWSTemplateFormatVersion: 2010-09-09
Description: EC2 instance
Resources:
  MyEC2Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: ami-0ff8a91507f77f867
      InstanceType: t2.micro
```

Code Example 1. EC2 instance with CloudFormation.

Code Example 1 demonstrates a CloudFormation template written in YAML, which creates an EC2 instance, with two properties specified ImageId and InstanceType. ImageId configures the virtual machines operating system and other installed software while InstanceType determines how much CPU, memory, and storage capacity the machine will have.

Campbell (2020) mentions some drawbacks for using CloudFormation:

- Service support availability: It takes time before support for new services becomes available for CloudFormation. Early adopters are better suited with third-party tools which provide support for new services more quickly.
- Limitations to extensibility: Custom logic is relegated to only Lambda functions, if local logic is needed it must be done through a wrapper script or program. Other IaC tools like Terraform and Sceptre provides interfaces and hooks for local operations.
- Lack of expressiveness in domain-specific language: Template logic is not extensible and many general-purpose logic mechanisms are not supported such as loops. In Terraform you get access to looplike constructs via its count metaparameter.

- Inconsistent experiences: The experience of deploying with local templates through the AWS Command-line Interface (CLI) is more restricted compared to staging the templates in an S3 bucket. The maximum size of templates is smaller and lacks access to features like Transform functions.
- Feature tradeoffs: Features such as StackSets promise improvements to functionality of stacks by allowing users to update multiple accounts and regions with a single operation. But by using StackSets users are then forced to give up access to features like Transform functions which have variety of uses within templates.

Overall AWS CloudFormation is a powerful tool for automating the process of deploying and managing infrastructure in the AWS Cloud. With its support for infrastructure as code, stack management, and automated updates, it provides a simple way for users to manage their infrastructure and ensure that it is always in the desired state. Whether setting up infrastructure for a new application, automating deployments, or managing existing infrastructure, CloudFormation provides a flexible and scalable solution for AWS users.

4.2. AWS CDK

AWS Cloud Development Kit (AWS CDK) is an open-source software development framework for defining cloud infrastructure with programming languages and then deploying it through AWS CloudFormation (AWS CDK documentation, 2023). It allows developers to define their infrastructure using familiar programming languages, such as Java, TypeScript, and Python, instead of JSON or YAML templates used in traditional CloudFormation. It was launched in July 2019 as a code-first approach to defining cloud application infrastructure.

AWS CDK beings to bridge the gap between dev and ops tooling, as developers with skills across variety of languages can write reusable, component-based infrastructure without the need to learn a new tool. Developers only need to learn to work with new libraries and classes, something that happens in most development projects anyways. (Campbell, 2020)

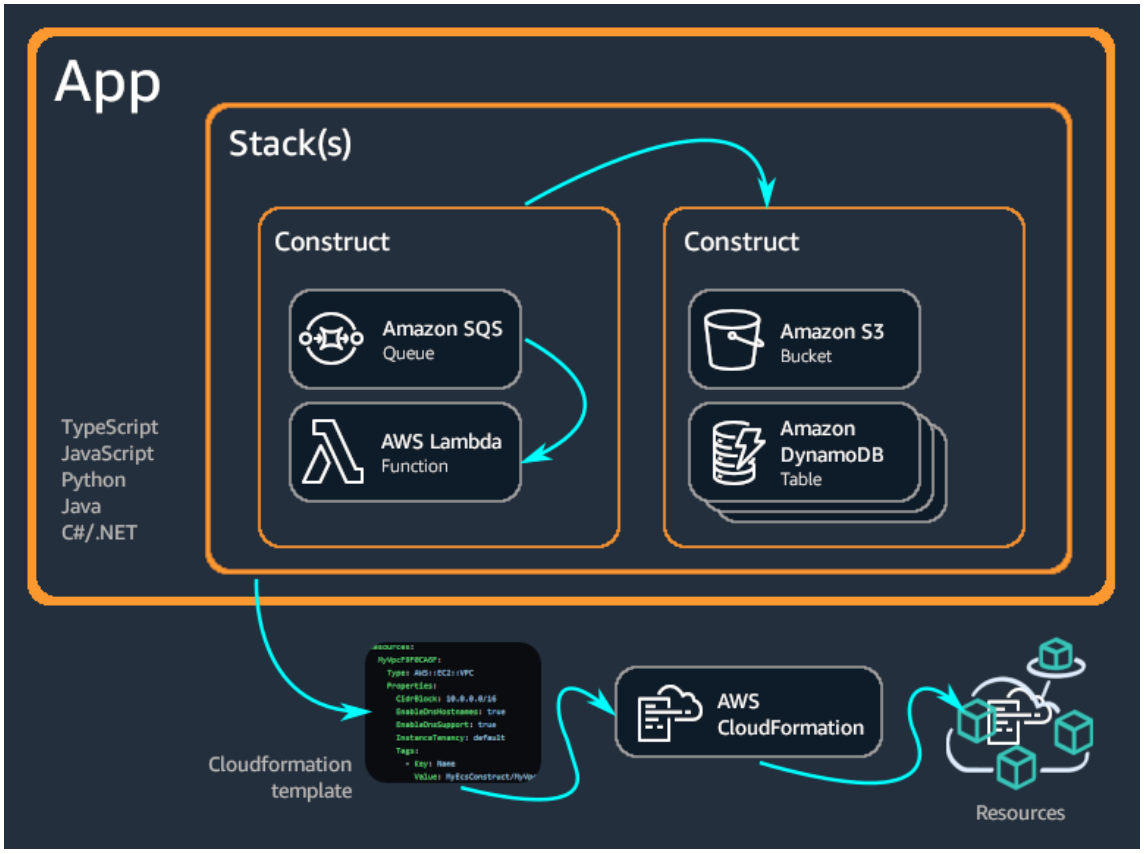

Figure 7. AWS CDK composition (AWS CDK documentation, 2023).

Seen from Figure 7 AWS CDK is composed of reusable cloud components known as constructs. Those are then composed together into stacks and apps. Constructs can represent a single AWS resource, or it can also be a higher-level abstraction consisting of multiple related resources. Such as worker queue with its associated compute capacity or scheduled job with monitoring resources and a dashboard (AWS CDK documentation, 2023).

Code Example 2 illustrates the creation of an EC2 instance with two different constructs using AWS CDK with TypeScript. First one, ec2Instance is created using the CfnInstance construct, which represents a CloudFormation EC2 instance. The other ec2Instancev2 is created using a higher-level construct called Instance. These higher-level constructs aim to simplify the process of defining EC2 instances and associated resources by providing opinionated defaults, encapsulating best practices, and abstracting away some of the low-level details. AWS CDK still allows for using the familiar CloudFormation resources if needed.

```
import * as ec2 from "aws-cdk-lib/aws-ec2";

    const ec2Instance = new ec2.CfnInstance(this, "Cfn EC2Instance", {
      imageId: "ami-0ff8a91507f77f867",
      instanceType: "t2.micro",
    });

    const ec2Instancev2 = new ec2.Instance(this, "CDK EC2Instance", {
      vpc: ec2.Vpc.fromLookup(this, "VPC", { isDefault: true }),
      instanceType: new ec2.InstanceType("t2.micro"),
      machineImage: ec2.MachineImage.latestAmazonLinux(),
    });
```

Code Example 2. EC2 instance with AWS CDK.

For allowing the use of multiple programming languages, AWS CDK uses JSii. It is a build tool that allows for code in any language to interact with JavaScript classes. The source module of AWS CDK which is written in TypeScript, is compiled to JavaScript using the JSii compiler, which produces a language-independent representation of the class library APIs. The JSii package manager then packages it as a module for all the supported languages. This allows for the library to be used by a variety of different programming languages, without having to write separate code for each language. Currently AWS CDK supports TypeScript, JavaScript, Python, Java, C# and Go. (Marcadier, 2020)

4.3. Terraform

Terraform is an open-source tool created by HashiCorp that allows you to define your infrastructure as code using a simple, declarative language and to deploy and manage that infrastructure across a variety of public cloud providers such as AWS and Azure and private cloud and virtualization platforms (Birkman, 2022). One of the big advantages of using Terraform is the ability to use it with a variety of cloud platforms. Terraform uses a custom DSL called HashiCorp Configuration Language (HCL) for its templates, which is a superset of JSON (Campbell, 2020).

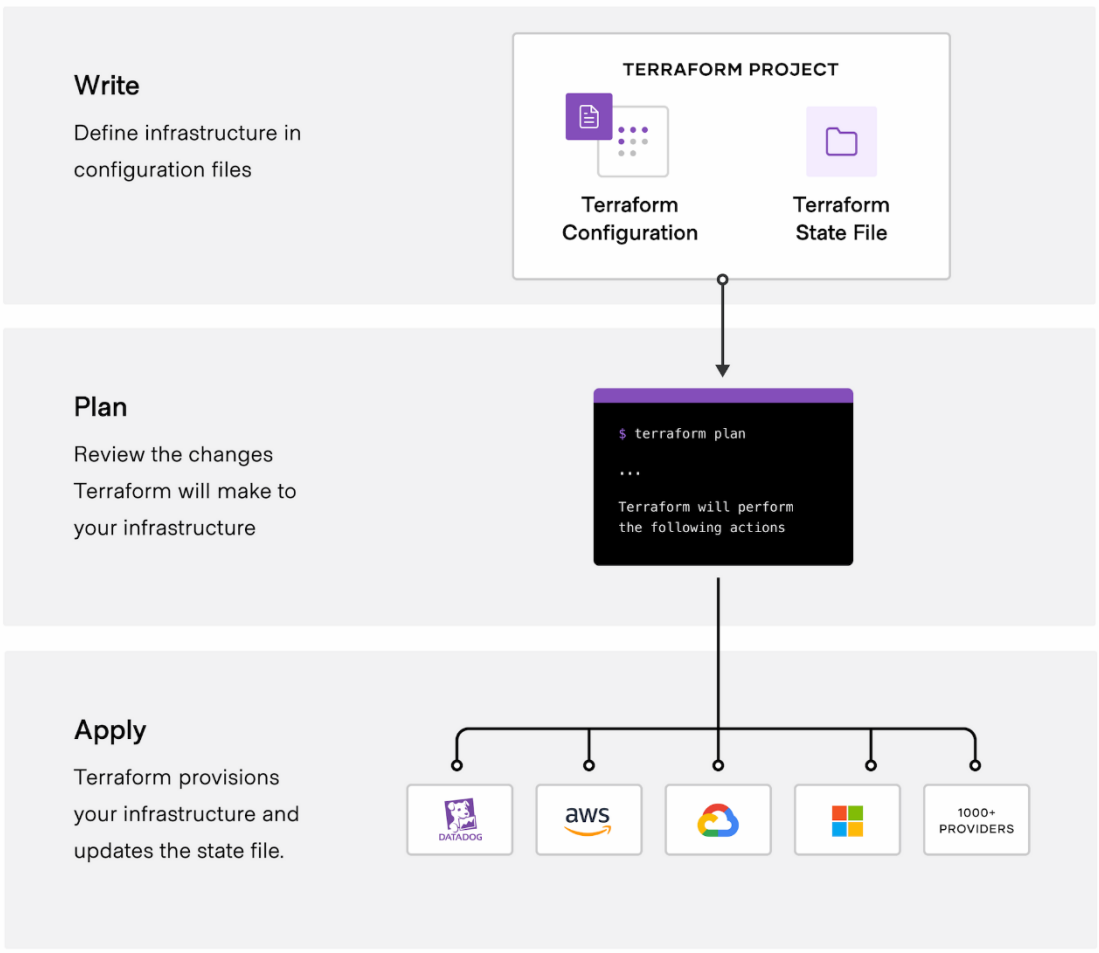

Figure 8. Terraform workflow (Terraform documentation, 2023).

Figure 8 breaks down the Terraform workflow into three stages: write, plan and apply. First resources need to be defined, which may be across multiple cloud providers and services. Then in the plan stage Terraform creates an execution plan describing the infrastructure it will create, update or destroy. Once the plan is approved Terraform performs the purposed operations to provision the infrastructure and updates the state file. (Terraform documentation, 2023).

Terraform providers are a logical abstraction of an upstream API. Each provider has its own set of resources, which represent different infrastructure objects, such as servers, networks, and storage, that can be managed using Terraform. For the AWS provider Terraform uses the AWS SDK for its interactions with the AWS API layer (Campbell, 2020). The provider's resources are defined in Terraform configurations, and users can create, modify, and destroy these resources using the Terraform CLI, which manages the under-

lying cloud provider APIs on behalf of the user. In the Terraform registry there are currently 35 providers that are maintained by HashiCorp and close to 3000 providers when counting partner and community providers (Terraform documentation, 2023).

```
resource "aws_instance" "ec2_instance" {
  ami           = "ami-0ff8a91507f77f867"
  instance_type = "t2.micro"
}
```

Code Example 3. EC2 instance with Terraform.

Code Example 3 shows how a EC2 instance is created with Terraform. The configuration above represents the desired state of a single EC2 instance and Terraform will create or update the instance accordingly. The “aws_instance” resource is provided by the AWS Terraform provider.

In 2020 Terraform launched their own cloud development kit similar to AWS CDK called CDKTF. It allows users to define infrastructure with familiar programming languages such as TypeScript, Python, Java and Go. CDKT uses the same JSii tool as AWS CDK to support multiple languages from a single TypeScript codebase. Instead of synthesizing to CloudFormation, CDKTF synthesizes to Terraform config files that are then deployed. Hashicorp recommends using CDKTF when users want to create abstractions to manage complexity but warn of breaking changes as the tool has not had its initial release yet. (Terraform documentation, 2023)

5. AWS CDK vs Terraform

This chapter is about the architecture, development, and experience of implementing the case study for comparing AWS CDK and Terraform. First in the background chapter we look at the objectives, restrictions, and limitations regarding the study. The architecture chapter explains the AWS infrastructure designed for the study. The implementation chapter goes through the steps and methodologies used during the implementation phase for both the IaC tools.

5.1. Background

This case study compares two IaC tools for deploying and managing cloud infrastructure on AWS. AWS CDK is an AWS-specific tool that allows defining resources using familiar programming languages. The other tool compared is Terraform, which is a cloud-agnostic tool that uses declarative language to describe infrastructure. These two tools were chosen because they were both popular tools among cloud practitioners and worked in different ways. Our prior knowledge of both the tools was low and of cloud development in general. We have done development with TypeScript and JavaScript before, which could be helpful with learning to use AWS CDK.

The objective is to build an identical AWS infrastructure with both tools and in the process study how the tools work and perform. Points of comparison include performance, maintainability, and developer experience. Another objective is to during the development try to get a picture of the different tools available and community support provided.

The case study will only use the AWS services that are included in the AWS free tier, which will limit what services can be used. AWS free tier includes free trials, limited time offers and some services which are always free. The architecture and the applications running in AWS are not the focus of this study. The main objective is to observe the differences in solving the same problems between the two IAC tools and how they perform. The application architecture uses established cloud architecture patterns to mimic real infrastructure on a smaller scale.

5.2. Architecture

To test the two IaC tools an example application needed to be built. A simple React (2023) application was created that allowed users to write messages that were then listed on the website. The application is split into two parts, a static website frontend with the React application served by CloudFront and the serverless backend, leveraging API Gateway, Lambda functions and DynamoDB as a database (AWS API Gateway, 2023; AWS CloudFront, 2023; AWS DynamoDB, 2023; AWS Lambda, 2023). The example application's architecture can be observed from Figure 9 below.

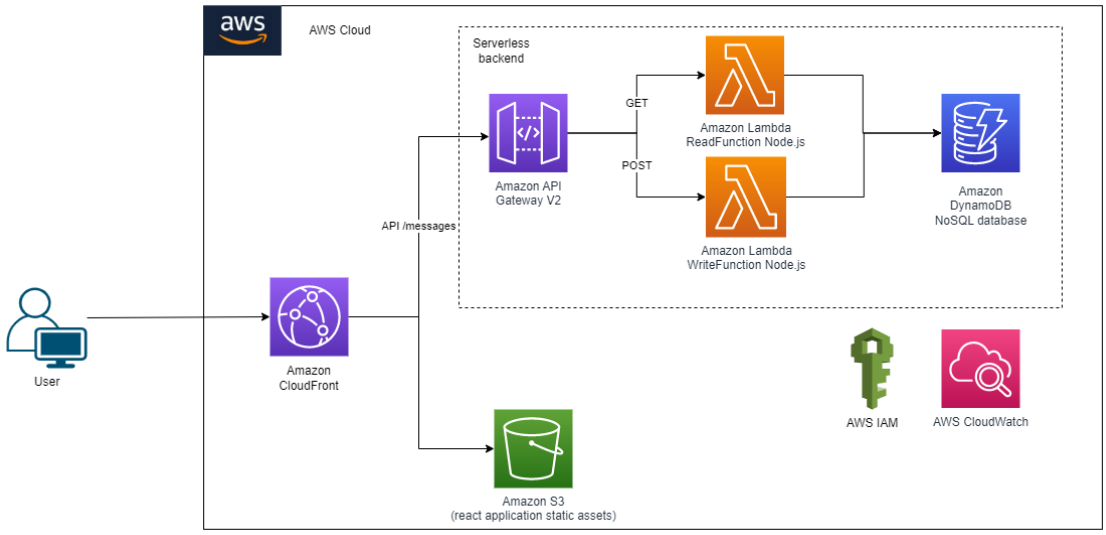

Figure 9. Example application architecture.

The static website is served by CloudFront which is a content delivery network (CDN) that speeds up distribution of static and dynamic web content, such as .html, .css and .js files (AWS CloudFront, 2023). CloudFront delivers content through a worldwide network of data centers called edge locations. When a user requests the web page, the request is routed to the edge location that provides the lowest time delay to deliver the content as fast as possible. CloudFront needs an origin that contains the content of the web page, in our case a S3 bucket with the React application static assets. S3 short for Simple storage service is a scalable object storage. S3 allows for storing and retrieving any amount of data. To allow CloudFront to retrieve the data from S3, an AWS Identity and Access Management policy (IAM) needs to be generated (AWS IAM, 2023). AWS IAM policies define permissions for an action for a specific resource.

The backend is built using serverless architecture. Serverless computing leaves server management to the cloud provider, allowing the developer to focus on the application logic and leaves the management of scaling and hosting of resources to the cloud provider. Serverless started from just function as a service (FaaS) to a large ecosystem of exclusive serverless components. AWS has a wide range of serverless services from Lambda and Fargate providing compute, API Gateway, SNS and SQS for application integration and serverless datastores like S3 and DynamoDB. (AWS SNS, 2023; AWS SQS, 2023; Paul, 2023)

API Gateway is a fully managed service that works as a link between different backend services. We are using API Gateway as a REST API for the application's backend. Through API Gateway user can define routes, authorization, define API keys and monitor API calls through dashboards. In the project API Gateway there is two routes for /messages, one for GET and one for POST HTTP-requests. Each route invokes its own Lambda function that then either writes a message to the DynamoDB database or gets messages from the database. (AWS API Gateway, 2023)

Lambda is a FaaS service with a pay-for-use pricing model, user focuses on writing the functions and AWS handles the underlying infrastructure. Lambda is a core AWS service, that integrates with most AWS Services. Lambda supports multiple languages through the use of runtimes. Runtimes provide language-specific environments that relay invocation events, context information and responses. In the project the functions are written in TypeScript and they're run using the Lambda Node.js 14.x runtime. (Paul, 2023, AWS Documentation 2023)

Amazon CloudWatch allows for monitoring resources and applications on AWS (AWS CloudWatch, 2023). It works through collecting metrics and logs from cloud resources and then displaying them through dashboards. For the example application we are collecting CloudWatch logs for the Lambda function executions and access logs for API Gateway requests.

5.3. Implementation

In this chapter we share the experience of learning the tools, the development process and any difficulties encountered along the way. To try and implement both infrastructures as close to identical as possible we did them in parts. With first implementing the static site frontend infrastructure with Terraform and then replicating it with AWS CDK. Then we did the opposite and developed the serverless backend with AWS CDK first and then tried to replicate it with Terraform.

Source code for the case study implementation can be found on GitHub (Project Repository, 2023). The project is licensed under the MIT License, which means that the software is released under an open-source license that allows anyone to use, modify, and distribute the software for any purpose, including commercial purposes.

5.3.1. Installation of the tools and first deployment

Prerequisites for both the tools was installing the AWS CLI and having an AWS account and associated credentials. For AWS CDK Node.js was also required. For TypeScript development with AWS CDK, TypeScript needed to be installed through npm (Node Package Manager). For deploying infrastructure bootstrapping was also needed. Bootstrapping is the process of provisioning resources for AWS CDK before apps can be deployed (AWS CDK documentation, 2023). Then we did a tutorial for launching your first application with the official documentation AWS CDK documentation. The tutorial went through creating an application from a template, adding code to create an S3 bucket and explained different cdk commands such as *synthesize*, *diff* and *deploy*.

For Terraform Windows installation, only step was to download the Terraform binary and then adding it as a PATH variable. Then I completed an AWS tutorial from Terraform for deploying a EC2 instance. The tutorial explained different commands and Terraform keywords as the tutorial went on. Every step of the tutorial was clearly explained with a video walkthrough along with text information.

5.3.2. Starting the project

To initialize an AWS CDK project, we need to run *cdk init app –language typescript* in an empty directory. This creates an empty cdk project and installs the aws-cdk module library and its dependencies. AWS CDK creates bin, lib and test folders with sample boilerplate to get started with development. Figure 10 shows all the files and folders created with the *cdk init* command. The setup is very similar to create-react-app that helped popularize React as a frontend framework.

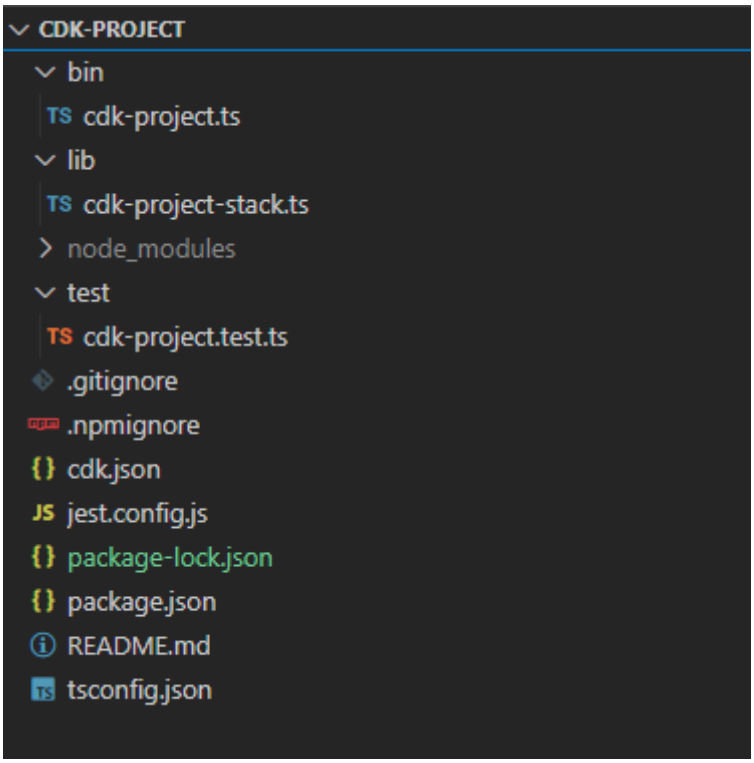

Figure 10. Files created with cdk init.

The bin folder contains the entry point for the CDK application. Code Example 4 below shows the contents of the project entry point. It includes the creation of the cdk app and then creates an instance of the AwsCdkStack class and deploys it using the CDK app. The second argument "AwsCdkStack" is the stack name. The env property specifies the AWS account and region to use, which are taken from environment variables. The lib folder contains the core of the implementation. In the file aws-cdk-stack.ts the stack is defined and the constructs: serverless-backend and static-site are created.

```
#!/usr/bin/env node
import "source-map-support/register";
import * as cdk from "aws-cdk-lib";
import { AwsCdkStack } from "../lib/aws-cdk-stack";

const app = new cdk.App();
new AwsCdkStack(app, "AwsCdkStack", {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION,
  },
});
```

Code Example 4. Entry point for the AWS CDK application.

With Terraform there was no ready-made project template generated like with AWS CDK. Everything need to be specified by the user from scratch. First step was defining the providers used for the project, which in our case was the AWS provider. Then we could run the command *terraform init*, which downloads the provider and installs them into a hidden subfolder. Provider configurations for the application shown in Code Example 5, source address for the provider and version are included.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.58.0"
    }
  }

  required_version = ">= 1.2.0"
}

provider "aws" {
  region = var.region
}
```

Code Example 5. Terraform provider definition.

Terraform modules were used to split the backend and frontend parts of the infrastructure implementation. Modules are containers for multiple resources that are used together, a module consist of multiple .tf and/or .tf.json files kept together in a directory.

Every Terraform configuration consist of at least one module known as the root module. The root module usually calls the other modules inside it. When a module is called by another module its referred as a child module. The static-site and serverless-backend are child modules called by the root module in environment directory. Modules help simplify Terraform code by abstracting away complex infrastructure details and making it easier to reuse common infrastructure patterns across different environments. (Terraform documentation, 2023).

5.3.3. Static site

For the static site portion of the project, we wanted to host a static React application. The React app implementation can be found in the react-app folder. We store our build files in the S3 bucket and then serve them through CloudFront. With Terraform this required creating an S3 bucket, adding it resources such as access control list, versioning, and public access block. Then creating an IAM policy for CloudFront for accessing the bucket and attaching it to a bucket policy. Implementation can be found in the folder /terraform/modules/static-site. With the AWS CDK implementation we created a new construct and then created the necessary constructs inside it. The implementation can be found in the /aws-cdk/lib/static-site.ts file.

5.3.4. Serverless backend

For the backend we developed the two Lambda functions with TypeScript and the database schema model using OneTable (2023). DynamoDB OneTable is a npm library for DynamoDB applications that use single-table design pattern. First function added a new message to the table and the other fetched messages back from the database. Implementation for them can be found from the lambda folder. The backend implementation was first developed using AWS CDK. First, we defined a new DynamoDB table and two Node.js Lambda functions. The table construct needed to grant write and read access to the Lambdas. Then we defined the API Gateway with two different routes, one for GET-requests and one for POST-requests. Finally, the two routes needed to be integrated with the Lambda functions. The AWS CDK implementation can be found from the /aws-cdk/lib/serverless-backend.ts file and the Terraform implementation from /terraform/modules/serverless-backend/ directory.

5.3.5. Testing

Jest (2023) came included with the application initialization for AWS CDK. Jest is a JavaScript testing framework. The AWS CDK documentation instructed to develop two categories of tests for AWS CDK apps: fine-grained assertions and snapshot tests. Assertion tests look for specific aspects of the generated CloudFormation template for correct values. Snapshot tests test the synthesized CloudFormation template against the previously stored template. They are meant to help with refactoring as you can track all the changes made to the template. For the project we implemented unit tests for both constructs and a snapshot test testing the whole stack. The tests can be found in the /aws-cdk/test/ folder. (AWS CDK documentation, 2023)

Terraform contains an experimental *test* command that allows building module acceptance tests. This is still in early development and features can change significantly, so we decided to use Terratest (Terraform documentation, 2023; Terratest, 2023). It is a library written in Go for testing infrastructure code including tools like Terraform, Docker and Kubernetes. It provides helper functions for working with cloud providers APIs, making HTTP requests and running shell commands (Terratest, 2023). It was used to create integration tests for both of the modules. The implemented tests can be found in the /terraform/test/ folder. Both tests first deploy the infrastructure and then check that the infrastructure is working correctly. In the serverless-backend test we first deploy the module, then check that the API Gateway works, then we send a new message through the gateway and finally check that the database has saved our message, finally we destroy the infrastructure.

5.3.6. Continuous Integration

CI pipelines are very common in software projects, so we wanted to try how both tools integrate with them. Both tools had great support for ready-made actions for running static code analysis tools, tests or for running Terraform or AWS CDK commands. The pipelines run all the static analysis and testing tools on the code and check for errors. Pipeline implementations can be found in the source repository from /.github/workflows folder.

6. Comparison

This chapter is about comparing the two tools selected regarding different software quality characteristics. The implementation discussed in Chapter 5 will be used as a base for conducting these comparisons. Main areas of comparison are performance, developer experience and maintainability.

6.1. Performance

Performance can be a key factor when choosing what IaC tool to use, as along the way longer deployment times can add up. With the nearly identical infrastructures created with AWS CDK and Terraform, we can compare which tool is faster. We used a Windows PowerShell CLI utility tool called Measure-Command, to measure the time it took to execute different operations (Powershell documentation, 2023). We measured the time it took in seconds to deploy the infrastructure, make a change to the infrastructure, and then the time it took to destroy the infrastructure. The same experiment was conducted 5 times with both the tools to take in account variance, ensure consistency, and accuracy of the results. The tests were conducted on a PC running Windows 10, Terraform version used for the comparison was 1.3.9 and the AWS CDK version used was 2.75.1.

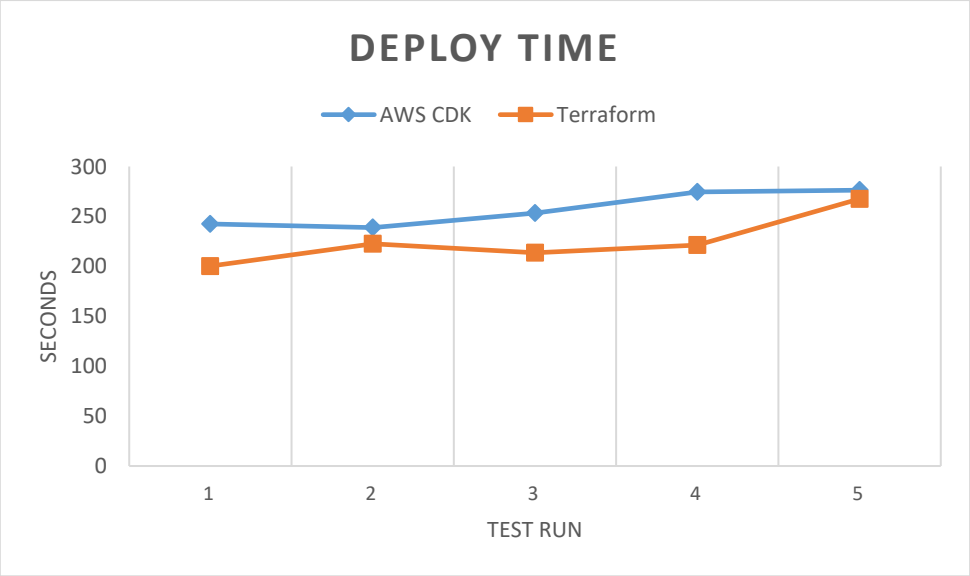

Figure 11. Deployment test results.

Terraform had faster deployment times, as illustrated in Figure 11. The deployment test took on average 257,1 seconds for AWS CDK and 225,06 seconds for Terraform. Deploying the CloudFront distribution took the most time for both tools.

For the infrastructure update test, we changed the Node.js version on the two Lambda functions from version 14 to 16. For AWS CDK it took on average 48 seconds to execute the update operation, with around 10 seconds of the time being used on synthesizing the code into a CloudFormation template. Terraform update operation took half the time averaging at 23,9 seconds. Results for each update execution run can be seen in the Table 1 below.

IaC Tool	Run 1	Run 2	Run 3	Run 4	Run 5	Average
AWS CDK	48,9	49,2	49,8	48,8	43,7	48,08
Terraform	23,5	24,1	24,6	23,9	23,6	23,94

Table 1. Infrastructure update test results.

The destroy results can be seen from Figure 12 below. The destroy time for both tools were similar with AWS CDK even reaching lower times than Terraform. On average it took AWS CDK 241,46 seconds to destroy the infrastructure and Terraform taking on average 222,96 seconds. Shortest destroy time was 207,9 seconds by Terraform while AWS CDK was close second with 209,7 seconds. With the destroy operation AWS CDK does not have to synthesize the code to CloudFormation which speeds up the destroy process.

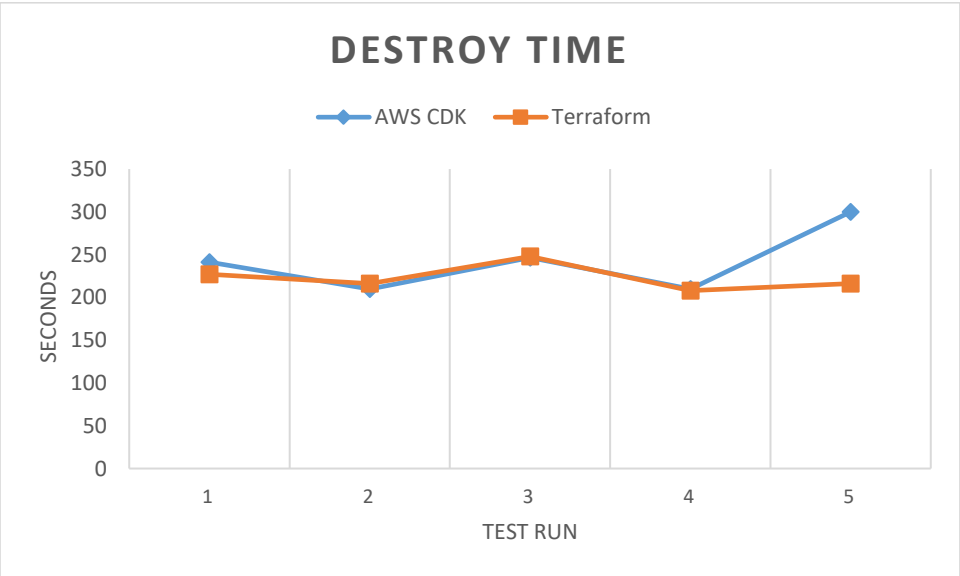

Figure 12. Destroy test results.

Based on the results collected, Terraform performed better than AWS CDK for all three operations, with lower average deployment, update, and destroy times. Terraform uses the AWS APIs for creating resources which helps it create infrastructure quickly. A contributing factor is also AWS CDK having to first synthesize the code. Synthesizing is a necessary step for deploying or updating infrastructure using AWS CDK, so it cannot be skipped or avoided. Synthesize duration varied between runs and operation but hovered around 10 seconds.

While the results of the experiment comparing Terraform and AWS CDK can provide some valuable insights into their performance differences, it is important to note that the experiment did not test for all scenarios. For example, the experiment only tested the performance of these tools with a limited set of resources and services found on AWS. It did not test how these tools perform when managing larger or more complex infrastructures, which could impact the performance results.

6.2. Developer Experience

We are focusing on evaluating the developer experience of using the tools, while still studying some of the same quality attributes found in usability. To compare developer experience between the two tools, we are focusing on the cognitive factors from the DX framework. Key question to answer: “How developers perceive their development infrastructure?”. This will be done through examining the learnability of the tools, IDE support available and the community support around the tools.

6.2.1. Learnability

We are measuring learnability by how easy it was to learn the tools and how easy it was to perform the tasks required for the project. The syntax of both tools is compared to see how they accomplished the same task. Initial learnability was measured by how easy the tool were to learn. Grossman et al. (2009) gathered four dimensions to consider in users when measuring initial learnability: level of experience with computers, level of experience with the interface, quality of domain knowledge and experience with similar software. Our level of experience with computers and the interface (in this case the IDE) was at expert level, while the domain knowledge and experience with similar software was on

beginner level. HCL and the use of declarative configuration languages was new, while we had about 3 years of previous experience with TypeScript.

Both tools were easy to install and get started with. The tutorials provided for Terraform guided through the installation process, basic commands, and concepts. AWS CDK had a similar tutorial for learning the commands and how the tool worked. Initial learnability of both tools was seamless, tutorials were quick and taught the main concepts well. Where AWS CDK shined was by guiding the use of best practices with the project initialization. With providing a project template with clear project structure and preinstalled test framework, allowed us to focus on creating the infrastructure. With Terraform for example use of modules was recommended, but not required. As there was no ready-made project template in place, it took longer to figure out.

The Terraform code was written in HCL, which is similar to JSON. It's a declarative language where you describe the infrastructure you want and Terraform figures how to create it (Birkman, 2022). The AWS CDK code was written in TypeScript, which is a strongly typed programming language that builds on JavaScript. TypeScript supports tight integration with code editors allowing to catch errors early and uses type inference for great tooling. It was the 4th most loved and wanted programming language in the 2022 Stack Overflow Developer survey. This yearly survey examines all aspects of the developer experience from learning to code to their favorite technologies to version control and the workplace experience of professional developers. (Stack Overflow Developer Survey, 2022)

```
// AWS CDK
table.grantReadData(readFunction);

// Terraform
resource "aws_iam_policy" "read_dynamodb_policy" {
  name = "read_dynamodb_policy_${var.env}"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Action = [
          "dynamodb:Query",
        ]
        Resource = [
          aws_dynamodb_table.messages_table.arn
        ]
      }
    ]
  })
}

resource "aws_iam_role" "iam_for_lambda_read" {
  name               = "iam_for_lambda_read_${var.env}"
  assume_role_policy = data.aws_iam_policy_document.assume_role.json
}

resource "aws_iam_role_policy_attachment" "read_lambda_attachment" {
  role       = aws_iam_role.iam_for_lambda_read.name
  policy_arn = aws_iam_policy.read_dynamodb_policy.arn
}
```

Code Example 6. Granting read access to a DynamoDB table with both tools.

With the abstraction layer provided by constructs AWS CDK has the ability to do things with less code. In Code Example 6 above, there is two code snippets from the project for granting a Lambda function read access to a DynamoDB table. With AWS CDK we only need to write one line for that. The table variable is a DynamoDB table construct which has a function called *grantReadData* that takes the Lambda construct as a parameter. While with Terraform, we had to define the read policy, role and then the role policy attachment to accomplish the same task. With having two Lambda functions, one requiring read access and one with write access, we had to repeat the almost identical

code twice. This leads to a lot of repetitive code and in a bigger project could cause opportunities for errors and misconfigurations.

Learnability wise both tools were easy to install and get started with. AWS CDK with the use of familiar programming languages offered an easier way for us to define infrastructure. AWS CDK is the easier tool to use, especially if the developer does not have much experience with working with AWS, as the abstraction layer helps simplify infrastructure definition. Terraform on the other hand uses a declarative style, which can give a clearer view of all the infrastructure created. If the developer does not have experience with any of the languages supported by AWS CDK, Terraform can be easier to learn.

6.2.2. IDE Support

For the IDE to develop the example project Visual Studio Code (2023) was chosen. It's a source-code editor from Microsoft, that has been gaining in popularity in recent years. It is main demographic is JavaScript and web developers, but with a plethora of extensions it can be tailored to fit different needs. We wanted to use one IDE for developing everything in the project including the IaC tools, React, Node.js and GitHub Actions.

We used the HashiCorp Terraform extension for developing the Terraform side of the infrastructure. It features included intellisense, syntax validation, syntax highlighting, code navigation and formatting. The extension worked well; we especially liked the code navigation feature. With Terraform code being split into different modules and files, the ability to jump between files helped a lot. Code validation worked, but we would have liked to get an error message when defining resources without a required property. These errors were only found on when running terraform plan or deploy. There were lots of other extensions available, the keyword "Terraform" returned 76 extensions from the VS Code extensions marketplace. Microsoft has even developed a separate Azure Terraform extension for supporting Terraform development on Azure.

For AWS CDK, there were not many extensions available. Searching the marketplace with the keyword "AWS CDK", only returned 3 results. There were some third-party extensions that added CDK L1 construct code snippets. As we are using TypeScript, there are a lot of TypeScript extensions available. VS Code also comes with TypeScript language support included. This meant most of the same features were available as in Ter-

raform, such as code highlighting, validation, and jumping between files. Especially useful was the ability to delve deeper into the source code of the aws-cdk module library to see what properties different constructs could take.

With Terraform we had to use the documentation for each resource to look for their properties. It made development much easier when you could check straight in the IDE what properties constructs can take, compared to browsing the internet for documentation and examples. Working with Terraform you are expected to use the provider registry documentation while developing and the IDE works mainly for formatting code and jumping between files. Due to typing the IDE can also warn me for misconfigurations, before deploying the infrastructure.

Overall, both tools had good IDE support for VS Code with several extensions available to provide a good development experience. TypeScript support allowed the IDE to provide more auto-suggestions and error prompts, together with the ability to view the source code of the constructs made us prefer development using AWS CDK.

6.2.3. Community support

The community support comparison includes looking at the resources available for both tools to help users learn, troubleshoot, and develop with the tool. It also includes factors such as quality of documentation, amount of third-party tooling and update frequency. Both tools are open-source which allows us to look at their GitHub repository metrics. Looking at the GitHub metrics collected in Table 2, we can see that both have garnered a lot of community support. Stars are a way for GitHub users to “like” a repository and is often used to measure how popular an open-source project is. A fork is a new repository that shares code and visibility settings from the original repository. They’re often used to iterate on ideas and used to purpose changes to the original repository. A large number of forks can indicate the existence of a vibrant community around the repository. It suggests that developers are actively engaged with the project, building upon it, and creating derivative works or variations to meet specific needs or use cases. Terraform is leading in most categories, but that can be explained by it being released 4 years earlier. Both tools are getting weekly updates, mainly in the form of bug fixes.

GitHub statistics	AWS CDK	Terraform
Stars	10100	36900
Forks	3200	8500
Contributors	1198	1715
Pull requests	71	166
Open Issues	1671	1643
Closed Issues	10350	18064
Release cycle	Weekly	Weekly
First commit	2018	2014

Table 2. GitHub statistics gathered in April 2023.

Documentation and guides used during the project:

- AWS CDK Developer Guide *https://docs.aws.amazon.com/cdk/v2/guide/home.html*
- AWS CDK Workshop *https://cdkworkshop.com/*
- AWS CDK Reference Documentation *https://docs.aws.amazon.com/cdk/api/v2/*
- Terraform Documentation and Guide *https://developer.hashicorp.com/terraform*
- Terraform Registry AWS Provider *https://registry.terraform.io/providers/hashicorp/aws/latest/docs*
- Various YouTube tutorials and guides found through Google

AWS CDK offers an API reference documentation, as well as a developer guide. During the project we did not have to use the API reference as much as the Terraform documentation, due to how much of the configuration is done behind the scenes already. Also, the fact that property defaults and options can be found through the IDE reduced the need to fall back on documentation. The reference documentation included examples, how to initialize and properties with typing and descriptions for each construct. Only thing we found lacking was a search bar to look for a specific construct or module. The developer guide helped at getting started with development, explained the key concepts, and provided tips for best practices and testing. AWS CDK provided an introduction CDK workshop available in different programming languages. (AWS CDK documentation, 2023)

Most used documentation during the development of the example project with Terraform was the Terraform Registry documentation for the AWS provider. With Terraform

all the different resources need to be configured manually, which required a lot of checking for required properties and example configurations for resources. For each resource the documentation provided example usage and the argument reference. Terraform also has a developer guide which includes a getting started tutorial and introduction to Terraform. The tutorials provided were comprehensive and easy to follow. (Terraform documentation, 2023)

With the Terraform implementation of the serverless backend we ran into some problems. We got the infrastructure deployed but something was wrong. Our API Gateway was giving an error message and not connecting to the Lambda functions. After a lot of trial and error, we found out our gateway had not deployed. It was missing a configuration that automatically deployed the API. This error was mainly due to us not knowing the intricacies of how API Gateway works, but the example given by the Terraform documentation did not help our case. With AWS CDK this setting was automatically enabled.

Both tools offered high quality documentation with good guides for developer to get started with and deepen their knowledge. Terraform had a wider range of guides available compared to AWS CDK. This could be explained by Terraform being the are more mature tool, with a longer history and strong community backing. While Terraform has stronger community support currently, for both tools, users can find answers to their questions, solutions to common problems, and guidance on best practices through community forums, discussion boards, and online resources.

6.3. Maintainability

Maintainability is hard to measure or compare, but in this chapter, we try to look at how testing can be performed on both tools, how easy it is to develop tests and how well they reveal faults. We use a static code analysis tool called Checkov (2023) to analyze both codebases and compare found errors. Static code analysis tools for formatting and linting code are also explored.

6.3.1. Testability

In the survey conducted by Guerriero et al. (2019) testability was mentioned as a common issue in IaC development. We wanted to study how both tools have tried to tackle this problem. We used Jest for testing AWS CDK and Terratest for testing Terraform. To note

there are other testing libraries available for both tools, but this comparison only focuses these two tools.

Software testability was defined by Garousi et al. (2019) as ease of testing and ease of revealing faults. In terms of ease of testing, Jest felt easier to test with than Terratest. This can be explained by multitude of factors. For one, we had previous experience with Jest, due to its prevalence in the JavaScript ecosystem. It is a common unit testing library for testing React, Angular and Node.js applications. Another reason was how easy and quick the tests were to make. This was due to the documentation and guides available. The tests work for confirming that AWS CDK synthesizes the correct configuration to CloudFormation. As the tests were unit tests that just asserted for configurations found in the template, they were very quick to execute. Jest offered possibility for snapshot testing as well, which received mixed opinions. We found it tedious having to update the snapshot test every time we made a small change to the infrastructure. But it did give us a better view of the changes made by the AWS CDK code added, by showing what additional resources were created in the CloudFormation template.

With Terratest the tests took longer to create, mainly due to having to learn a completely new library and programming language. Terratest is designed for integration testing, deploying the infrastructure, and then verifying it works correctly. The tests created took longer to execute because it had to deploy the infrastructure. This made test development slower as every time you ran the tests, there was 1-3 minutes spent deployment before the actual tests happened. With the ability to send http requests during a test, we could test the interaction between the API Gateway, Lambda and DynamoDB. The tests gave a better indication on whether the infrastructure deployed and worked correctly compared to the unit tests written for AWS CDK. During the creation of the integration tests, we wondered if the same functionality could be tested by normal frontend end to end tests, which would test the infrastructure in the process. While ease of testing was more on the side of Jest, we felt Terratest was the better tool for revealing real faults in the infrastructure.

Overall, it is hard to tell which style of testing or tool is more efficient or effective. Testing IaC tools is still in its infancy and there is no one right way to do it. Jest provides quick unit testing with a familiar framework; tests are easy to develop and execute. Terratest on the other hand, tests take more time develop but can reveal more faults and test the integration between infrastructure components.

6.3.2. Static code analysis

Static code analysis tools are widely used in the software development to detect and prevent potential security vulnerabilities, performance issues, and other code quality problems. These tools work by analyzing the code without executing it and providing feedback on potential issues that may exist. In this chapter we look at tools and libraries available for static code analysis for both the IaC tools.

Checkov is a static code analysis tool for scanning IaC files for misconfigurations that may cause security or compliance issues. It is implemented in Python and uses policies to check source files for security vulnerabilities. Checkov 2.0 uses graph-based policies for improved variable resolution and increased performance when scanning Terraform source files. These new policies are marked with the CHKV2 tag. Currently Checkov supports scanning of Terraform, CloudFormation, Azure Resource Manager, Serverless framework, Helm charts, Kubernetes and Docker files (Azure Resource Manager, 2023; Helm, 2023; Serverless, 2023). For AWS CDK Checkov has 163 policies to check against and over 1100 for Terraforms AWS resources. (Nordhausen, 2023; Checkov documentation, 2023)

With Checkov we can compare the Terraform files and AWS CDK files we have created against different policies. For Terraform Checkov scans all the .tf files we have created and for AWS CDK it will analyze the CloudFormation template generated by our AWS CDK code. Results are shown in the Table 3 below. Checkov ran 132 tests against our Terraform code, of which 96 passed and 36 failed. With AWS-CDK it ran 105 tests, with 82 passing and 23 failing. There were 22 unique failed tests with the Terraform files and 12 with the AWS CDK code. Failed tests were related to policies around Lambda function configuration, ensuring DynamoDB tables and CloudWatch logs are encrypted with KMS and S3 bucket settings. 10 errors were shared among both codebases. These included policies such as CKV_AWS_115: "Ensure that AWS Lambda function is configured for function-level concurrent execution limit" and CKV_AWS_18: "Ensure the S3 bucket has access logging enabled". Most frequent policy broken for AWS CDK was related to Lambda functions and it appeared 4 times in the source code, Terraforms most frequent error was related to CloudWatch encryption with 3 errors. Test passing rate was close with AWS CDK having a little higher rate with 78 %, but with less policies in total tested compared to Terraform. While most of the errors could be ignored, this is a good tool to help develop secure infrastructure. One conclusion to make from these results is

that with AWS CDK it is easier to make safer infrastructure as sane default settings are preconfigured behind the scenes. With Terraform everything gets configured manually by the user, which leaves higher chance for misconfiguration.

Checkov results	AWS CDK	Terraform
Total tests	105	132
Passed	82	96
Failed	23	36
Passing rate	78 %	72 %
Unique errors	12	22
Most frequent error	4	3

Table 3. Checkov results.

Another way to automatically check source code for programmatic and stylistic errors is through use of a linting tool. For Terraform we used TFLint (2023), which finds possible errors, warns about deprecated syntax and enforces best practices. In Figure 13 TFLint has found an issue regarding a missing version constraint for the archive provider. For AWS CDK we used ESLint (2023), which is a JavaScript linter used for identifying problems. For code formatting Terraform comes with the command *terraform fmt*, which can be used to automatically check and format the source code. For AWS CDK, Prettier (2023) worked well for enforcing a desired code style.

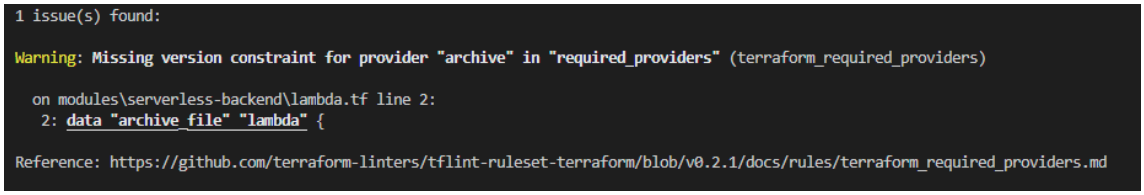

Figure 13. Issue found by TFLint in the Terraform codebase.

Both AWS CDK and Terraform have support for static code analysis to detect potential issues before deployment. AWS CDK has a built-in TypeScript/JavaScript compiler that can catch syntax errors, and it integrates with tools such as ESLint and Prettier for code quality checks. Terraform has a built-in syntax checker, formatter and it supports various third-party tools such as Checkov and TFLint for more advanced static analysis. Overall, regarding static code analysis tools, we feel Terraform ecosystem is more diverse and refined. This could be due to it being a more mature tool, more popular and has gotten better third-party support along the way.

6.4. Summary of the findings

Comparing the tools has given a lot of insight into their differences and similarities. Both tools have their own strengths and weaknesses. Table 4 summarizes the findings found for both the tools through the comparative study. Performance wise Terraform fared better in the tests, but not by a far margin. Terraform can interact directly with the underlying AWS API's for creating the infrastructure, while AWS CDK must first synthesize to CloudFormation which takes extra time. To get a better picture of the performance difference between the two tools more tests need to be performed, especially with larger scale infrastructures.

Quality	**AWS CDK**	**Terraform**
Performance	AWS CDK had to synthesize the code to CloudFormation which influenced the deployment and update times.	Terraform performed better in all the tests due to the ability to interact directly with the underlying AWS APIs.
Learnability	Easy to install and get started with. Abstraction layer reduced the complexity and amount of code required.	Quick to install and get started. DSL required time to get used to. Sometimes defining infrastructure required a lot of repetitive code.
IDE Support	Use of GPLs allows for better IDE integration and support, ability to get immediate feedback to errors from the IDE.	Good support available, with intellisense, code navigation and syntax highlighting.
Community Support	AWS CDK offers extensive and well-structured documentation, which serves as a valuable resource for users.	Popular tool with a lot of community made content with tutorials, blog posts and extensions.
Testability	Jest, unit testing and snapshot testing. Ease of testing: tests were quick to develop and quick to execute.	Terratest, integration testing. Tests took longer to implement and execute, but tests were better at revealing faults.
Static Code Analysis	Abstraction layer and sane default options reduce chance for misconfigurations. ESLint and Prettier for linting and formatting.	Everything is configured by the user, which leaves a chance for errors and mistakes. Built-in formatter, TFLint for linting.

Table 4. Summary of the findings.

Developer experience was studied through three factors: learnability, IDE support and community support. Both tools were easy to get started with, offered good tutorials and documentation. IDE support was tested using Visual Studio Code. Both tools had

intellisense, formatting, and code navigation. AWS CDK offered a better experience due to TypeScript warning off most errors, while most Terraform syntax and configuration errors were only revealed during deployment. AWS CDK allowed for the ability to view construct properties and default parameters from the IDE, reducing the amount off documentation reviewing needed during development.

Maintainability comparison focused on two factors: testability and static code analysis. Testability was measured by the ease of testing and the ease of revealing faults of the testing tools used. For AWS CDK Jest was used for unit tests and snapshot testing. The tests were easy to make and quick to execute. For Terraform Terratest was used for integration testing of the modules. The tests took longer to make and took longer to execute, due to requiring the infrastructure to be deployed and destroyed. Even though the tests were harder to make on Terraform, they felt more useful in revealing faults than the tests with Jest. Overall testing of infrastructure code felt still in it is infancy, a lot of ideas and different approaches. The static code analysis used an infrastructure code analysis tool called Checkov to analyze both codebases for issues and vulnerabilities. The AWS CDK codebase had less problems, which could indicate it is easier to make safe infrastructure with. The abstraction layer and best practice default values safeguard from misconfigurations. That said, the Terraform code did not have any glaring issues either. Linters and code formatters were found for both tools, Terraform had more specialized tools while AWS CDK took advantage of the JavaScript ecosystem.

In summary, both tools performed well in all categories. We favor AWS CDK for its imperative approach allowing the use of familiar programming languages. With our previous background of working with JavaScript, it was easier to get started and deploy infrastructure. The tool was more beginner friendly and worked well together with AWS. Terraform with its declarative style allows the developer more control and precision on defining the resources but required more time getting used to. In a project with a multi-cloud environment: for example, with AWS and Azure, is Terraform the clear option to allow handling all infrastructure code with one tool.

7. Conclusion

The aim of this thesis was to compare two infrastructure as code tools: Terraform and AWS CDK for managing cloud resources on Amazon Web Services. The comparative study aimed to evaluate the key features, functionality, and benefits of each tool, as well as their suitability for AWS development. The comparison was conducted through creating an example infrastructure that was then deployed using both IaC tools. With the help of the example project, we then compared the tools using different ISO/IEC 25010 software quality characteristics. Main areas compared were the performance of the tools, developer experience, and the support for maintainability.

In the performance comparison Terraform fared better, with faster deployments and update operations. AWS CDK was the easier tool to use and provided a better developer experience. Main reasons were better IDE support due to TypeScript typing and more concise syntax offered. Terraform has been around for a longer time and has gotten extensive community support. This can be seen through introduction of tools such as TFLint, Terratest and Checkov. Maintainability was compared through testability and static code analysis tooling. AWS CDK offered easier and faster testing with Jest but Terraform with Terratest allowed for more in-depth testing and was better at revealing faults in infrastructure. Both tools offered tools for linting and formatting code and had similar results in the static code analysis done.

We would recommend AWS CDK for anyone getting started with AWS and defining cloud infrastructure. The abstraction layer provided helps with managing complexity and reduces the amount of coded required. The ability to use a familiar programming language like TypeScript, combined with how it integrates with IDE support in Visual Studio Code were unmatched. AWS CDK guides the development process with their readymade project structure and examples, which lowered the barrier to entry. Getting started with development felt quick and easy. If we'd start a project that is exclusively focused around AWS, is AWS CDK the logical choice due to the better support and integration with AWS services.

While Terraform would be a better option for more experienced users, who know what kind of infrastructure they want and how to define them. It is a more stable and mature tool, that has been refined along the years and allows for use of the declarative programming with a domain-specific language. When deployment and update times for infrastructure matter is Terraform the better option. Terraform is a more flexible tool due

to the ability work with multiple cloud providers, which is a rising requirement in large software projects. Terraform leaves the door open for the change in cloud provider. While the infrastructure code will still need a rewrite, it does not vendor lock to using AWS.

Despite the comprehensive evaluation of AWS CDK and Terraform in this study, certain limitations should be acknowledged. Firstly, the study primarily focused on the AWS ecosystem, and the findings may not be directly applicable to other cloud platforms. Furthermore, the evaluation was conducted based on a specific set of criteria and scenarios, which may not encompass the entire range of possible use cases. Only a handful of AWS services were used, and the scale and complexity of the infrastructure was limited. Additionally, the comparison considered the current versions of both tools available at the time of the study, and future updates or enhancements may introduce new features and optimizations that could influence the overall assessment.

Regarding future work, the performance, portability, and scalability characteristics should be further studied. Conducting performance and scalability tests on large-scale infrastructure deployments could provide valuable insights into the tools' capabilities in real-world scenarios. Furthermore, investigating the integration of AWS CDK and Terraform with other DevOps tools and practices, such as continuous integration and delivery pipelines, could enhance the overall development and deployment workflows. Lastly, conducting user surveys or case studies to gather feedback and experiences from developers and teams using AWS CDK and Terraform in production environments would provide valuable insights into the practical implications and challenges of adopting these tools.

Ultimately, the decision to use Terraform or AWS CDK should be based on factors such as the complexity of the project infrastructure, the familiarity with programming languages, the level of integration required with AWS services, and the need for multi-cloud support. Both tools have their strengths and limitations, and the selection should align with the specific requirements and preferences of the development team.

References

Ansible. (2023). Infrastructure Automation Tool. https://www.ansible.com (Accessed: May 8, 2023)

AWS. (2023). Amazon Web Services, Inc. https://aws.amazon.com/about-aws/ (Accessed: February 2, 2023)

AWS API Gateway. (2023). Amazon Web Services, Inc. https://aws.amazon.com/api-gateway/ (Accessed: May 15, 2023)

AWS CDK documentation. (2023). Amazon Web Services, Inc. https://aws.amazon.com/cdk/ (Accessed: February 14, 2023)

AWS CloudFormation. (2023). Amazon Web Services, Inc. https://aws.amazon.com/cloudformation/ (Accessed: May 8, 2023)

AWS CloudFront. (2023). Amazon Web Services, Inc. https://aws.amazon.com/cloudfront/ (Accessed: May 12, 2023)

AWS CloudWatch. (2023). Amazon Web Services, Inc. https://aws.amazon.com/cloudwatch/ (Accessed: May 15, 2023)

AWS DynamoDB. (2023). Amazon Web Services, Inc. https://aws.amazon.com/dynamodb/ (Accessed: May 15, 2023)

AWS IAM. (2023). Amazon Web Services, Inc. https://aws.amazon.com/iam/ (Accessed: May 15, 2023)

AWS Lambda. (2023). Amazon Web Services, Inc. https://aws.amazon.com/lambda/ (Accessed: May 15, 2023)

AWS S3. (2023). Amazon Web Services, Inc. https://aws.amazon.com/s3/ (Accessed: May 15, 2023)

AWS SNS. (2023). Amazon Web Services, Inc. https://aws.amazon.com/sns/ (Accessed: May 15, 2023)

AWS SQS. (2023). Amazon Web Services, Inc. https://aws.amazon.com/sqs/ (Accessed: May 15, 2023)

Azure Resource Manager. (2023). Azure IaC Tool. https://azure.microsoft.com/en-us/get-started/azure-portal/resource-manager/ (Accessed: May 19, 2023)

Birkman, Y. (2022). *Terraform: Up and Running, 3rd Edition*. O'Reilly Media, Inc.

Campbell, B. (2020). *The Definitive Guide to AWS Infrastructure Automation Craft Infrastructure-as-Code Solutions* (1st ed. 2020.). Apress. https://doi.org/10.1007/978-1-4842-5398-4

Checkov. (2023). Static Code Analysis Tool for Infrastructure. https://www.checkov.io/ (Accessed: April 3, 2023)

Chef. (2023). Configuration Management Tool. https://www.chef.io/products/chef-infra (Accessed: May 8, 2023)

Cloudify. (2023). Cloud Orchestration Framework. https://cloudify.co/ (Accessed: May 8, 2023)

Coccia, M., & Benati, I. (2018). Comparative Studies. In A. Farazmand (Ed.), *Global Encyclopedia of Public Administration, Public Policy, and Governance* (pp. 1–7). Springer International Publishing. https://doi.org/10.1007/978-3-319-31816-5_1197-1

Dalla Palma, S., Di Nucci, D., Palomba, F., & Tamburri, D. A. (2020). Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, *170*, 110726. https://doi.org/10.1016/j.jss.2020.110726

Docker. (2023). Container Technology. https://www.docker.com/ (Accessed: May 8, 2023)

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *IEEE Software*, *33*(3), 94–100. https://doi.org/10.1109/MS.2016.68

ESLint. (2023). JavaScript Linter. https://eslint.org/ (Accessed: May 17, 2023)

Fagerholm, F., & Münch, J. (2012). Developer experience: Concept and definition. *Proceedings of the International Conference on Software and System Process*, 73–77.

Fife, C. (2018). *AWS CDK Developer Preview*. AWS Developer Tools Blog. https://aws.amazon.com/blogs/developer/aws-cdk-developer-preview/ (Accessed: March 2, 2023)

Freet, D., Agrawal, R., John, S., & Walker, J. J. (2015). Cloud forensics challenges from a service model standpoint: IaaS, PaaS and SaaS. *Proceedings of the 7th International Conference on Management of Computational and Collective IntElligence in Digital EcoSystems*, 148–155. https://doi.org/10.1145/2857218.2857253

Garousi, V., Felderer, M., & Kılıçaslan, F. N. (2019). A survey on software testability. *Information and Software Technology*, *108*, 35–64. https://doi.org/10.1016/j.infsof.2018.12.003

GitHub Actions. (2023). Continous Integration Platform. https://github.com/features/actions (Accessed: May 8, 2023)

Google App Engine. (2023). Google Cloud. https://cloud.google.com/appengine (Accessed: May 15, 2023)

Grossman, T., Fitzmaurice, G., & Attar, R. (2009). A survey of software learnability: Metrics, methodologies and guidelines. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 649–658. https://doi.org/10.1145/1518701.1518803

Guerriero, Mi., Garriga, M., Tamburri, D. A., & Palomba, F. (2019). Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 580–589. https://doi.org/10.1109/ICSME.2019.00092

Helm. (2023). Kubernetes Package Manager. https://helm.sh/ (Accessed: May 19, 2023)

ISO. (2023). ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (Accessed: February 28, 2023)

Jest. (2023). JavaScript Testing Framework. https://jestjs.io/ (Accessed: May 16, 2023)

Kavis, M. J. (2014). *Architecting the cloud: Design decisions for cloud computing service models (SaaS, PaaS, and IaaS)* (1st ed.). WILEY. https://doi.org/10.1002/9781118691779

Krishnan, S. (2010). *Programming Windows Azure* (First edition.). O'Reilly.

Kubernetes. (2023). Container Orchestration Platform. https://kubernetes.io/ (Accessed: May 8, 2023)

Kumara, I., Garriga, M., Romeu, A. U., Di Nucci, D., Palomba, F., Tamburri, D. A., & van den Heuvel, W.-J. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology*, *137*, 106593. https://doi.org/10.1016/j.infsof.2021.106593

Marcadier, R. (2020). *How the jsii open source framework meets developers where they are*. AWS Open Source Blog. https://aws.amazon.com/blogs/opensource/how-the-jsii-open-source-framework-meets-developers-where-they-are/ (Accessed: April 12, 2023)

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing—The business perspective. *Decision Support Systems*, *51*(1), 176–189. https://doi.org/10.1016/j.dss.2010.12.006

McCombes, S. (2019). *What Is a Case Study? | Definition, Examples & Methods*. Scribbr. https://www.scribbr.com/methodology/case-study/ (Accessed: April 27, 2023)

Morris, K. (2021). *Infrastructure as code: Dynamic systems for the cloud age* (2nd edition). O'Reilly Media, Incorporated.

Mustafa, C., & Zeebaree, S. (2021). *Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review*. *5*, 17–30. https://doi.org/10.5281/zenodo.4481415

Nordhausen, B. (2023). *Terraform for Google Cloud Essential Guide: Learn How to Provision Infrastructure in Google Cloud Securely and Efficiently.* (1st ed.). Packt Publishing, Limited.

OneTable. (2023). DynamoDB Access Library. https://github.com/sensedeep/dynamodb-onetable (Accessed: May 19, 2023)

Packer. (2023). Machine Image Automation Tool. https://www.packer.io/ (Accessed: May 8, 2023)

Paul, J. J. (2023). *Distributed Serverless Architectures on AWS: Design and Implement Serverless Architectures*. Apress L. P.

Powershell documentation. (2023). Microsoft PowerShell Utility. https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command (Accessed: April 17, 2023)

Prettier. (2023). Code Formatter. https://prettier.io/index.html (Accessed: May 17, 2023)

Project repository. (2023). GitHub. https://github.com/anttipessa/aws-cdk-terraform-comparison (Accessed: April 20, 2023)

Pulumi. (2023). Infrastructure as Code Tool. https://www.pulumi.com/ (Accessed: May 8, 2023)

Puppet. (2023). Infrastructure Automation Tool. https://www.puppet.com/ (Accessed: May 8, 2023)

Rajaraman, V. (2014). Cloud computing. *Resonance*, *19*(3), 242–258. https://doi.org/10.1007/s12045-014-0030-1

React. (2023). Frontend JavaScript Library. https://react.dev/ (Accessed: May 15, 2023)

Serverless. (2023). Infrastructure as Code Tool. http://serverless.com// (Accessed: May 19, 2023)

Srinivasan, V. (2018). *Google cloud platform for architects: Design and manage powerful cloud solutions* (1st edition). Packt.

Stack Overflow Developer Survey. (2022). Stack Overflow. https://survey.stackoverflow.co/2022/ (Accessed: May 11, 2023)

Surbiryala, J., & Rong, C. (2019). Cloud Computing: History and Overview. *2019 IEEE Cloud Summit*, 1–7. https://doi.org/10.1109/CloudSummit47114.2019.00007

Synergy Research Group. (2022). https://www.srgresearch.com/articles/q3-cloud-spending-up-over-11-billion-from-2021-despite-major-headwinds-google-increases-its-market-share (Accessed: February 3, 2023)

TechTarget. (2023). IT Encyclopedia. https://www.techtarget.com/searchsoftwarequality/definition/polyglot-programming (Accessed: May 2, 2023)

Terraform documentation. (2023). Terraform by HashiCorp. https://www.terraform.io/ (Accessed: February 20, 2023)

Terratest. (2023). Infrastructure Testing Tool. https://terratest.gruntwork.io/ (Accessed: May 16, 2023)

TFLint. (2023, April 19). Terraform Linter. https://github.com/terraform-linters/tflint (Accessed: April 19, 2023)

Visual Studio Code. (2023). Code Editor. https://code.visualstudio.com/ (Accessed: May 8, 2023)

Wang, R. (2022). *Essential Infrastructure as Code*. Manning Publications.

Warner, T. (2020). *Microsoft Azure For Dummies* (1st edition). John Wiley & Sons.