

BACHELOR THESIS

Alemseged Zerihun Demissie

Design and prototype implementation of an analytics platform in Azure using DevOps and Infrastructure as Code

FAKULTÄT TECHNIK UND INFORMATIK

Department Informations- und Elektrotechnik

FACULTY OF ENGINEERING AND COMPUTER SCIENCE

Department of Information and Electrical Engineering

Alemseged Zerihun Demissie

Design and prototype implementation of an analytics platform in Azure using DevOps and Infrastructure as Code

Bachelor Thesis based on the examination and study regulations for the Bachelor of Engineering degree programme Information Engineering at the Department of Information and Electrical Engineering of the Faculty of Engineering and Computer Science of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Kolja Eger
Second examiner: Prof. Dr Robert Heß
Day of delivery: 06. January 2023

Alemseged Zerihun Demissie

Title of Thesis

Design and prototype implementation of an analytics platform in Azure using DevOps and infrastructure as Code

Keywords

Cloud, DevOps, Azure, Open Data, Analytics, Prototype

Abstract

This paper details the design and prototype implementation of an analytics platform on Microsoft Azure, leveraging DevOps and Infrastructure as Code (IaC). The project integrates containerization and Azure services, utilizing Azure DevOps for continuous integration and Terraform for infrastructure management. The focus is on demonstrating the efficiency, scalability, and reliability of the platform in a cloud environment. The findings highlight the practicality of combining cloud computing, DevOps, and IaC, offering valuable insights for similar implementations in the cloud computing field.

Alemseged Zerihun Demissie

Thema der Bachelorthesis

Design und Prototyp-Implementierung einer Analyseplattform in Azure unter Verwendung von DevOps und Infrastruktur als Code

Stichworte

Cloud, DevOps, Azure, Open Data, Analytics, Prototyp

Kurzzusammenfassung

In diesem Dokument werden der Entwurf und die Prototypenimplementierung einer Analyseplattform auf Microsoft Azure unter Nutzung von DevOps und Infrastructure as Code (IaC) detailliert beschrieben. Das Projekt integriert Containerisierung und Azure-Dienste und nutzt Azure DevOps für die kontinuierliche Integration und Terraform für die Infrastrukturverwaltung. Der Fokus liegt auf dem Nachweis der Effizienz, Skalierbarkeit und Zuverlässigkeit der Plattform in einer Cloud-Umgebung. Die Ergebnisse unterstreichen die Praktikabilität der Kombination von Cloud Computing, DevOps und IaC und bieten wertvolle Erkenntnisse für ähnliche Implementierungen im Cloud-Computing-Bereich.

Table of Contents

1. INTRODUCTION	5
1.1 ENTOSE-E	5
1.2 TOPIC STRUCTURE	6
2. BACKGROUND	7
2.1 CLOUD.....	7
2.1.1 <i>How cloud computing functions</i>	7
2.1.2 <i>Cloud Infrastructure</i>	7
2.1.3 <i>Cloud Deployment Models</i>	8
2.1.4 <i>Cloud services</i>	8
2.1.5 <i>Azure Cloud Provider</i>	9
2.1.6 <i>Main service models of cloud computing</i>	9
2.2 INFRASTRUCTURE AS CODE	10
2.3 VIRTUAL MACHINES	11
2.4 CONTAINERIZATION	12
2.5 VIRTUALIZATION VS. CONTAINERIZATION.....	13
2.6 SERVERLESS	14
2.7 DEVOPS	16
2.7.1 <i>DevOps Architecture</i>	16
3. REQUIREMENTS.....	19
3.1 REQUIREMENT OVERVIEW	19
4. DESIGN.....	23
4.1 CONTAINERIZED WEB APP SERVICE	23
4.1.1 <i>Technologies and Tools</i>	24
4.1.2 <i>Architecture Design for Containerized Web App</i>	28
4.2 AZURE FUNCTIONS APP	29
4.2.1 <i>Architecture Design for Azure functions</i>	30
4.3 FRONTEND	32
5. IMPLEMENTATION	34
5.1 INITIAL CONFIGURATION AND LOCAL ENVIRONMENT SETUP	34
5.2 INFRASTRUCTURE.....	35
5.2.1 <i>Code Structure and Best Practices for Terraform</i>	35
5.2.2 <i>Database</i>	36
5.2.3 <i>Web App Backend</i>	36
5.2.4 <i>Serverless Azure Functions</i>	37
5.2.5 <i>Static Web App Frontend</i>	38
5.3 SOURCE CODE.....	39
5.3.1 <i>FastAPI Web Service</i>	39
5.3.2 <i>Serverless function app</i>	40
5.4 PIPELINES.....	40
6. TEST AND EVALUATION	42
7. SUMMARY.....	45
BIBLIOGRAPHY	46

1. Introduction

Cloud computing and DevOps enhance software development and deployment process. Cloud computing offers scalable resources and services on-demand, thereby removing the need for local infrastructure and facilitating seamless data and resource management across the internet. DevOps complements this by optimizing the development lifecycle, promoting continuous integration and deployment, with these technologies the primary objective of this thesis project is to design and develop a prototype for an analytics platform. Hosted on the Azure cloud, this platform aims to provide users with intuitive insights into electricity generation, interacting seamlessly with them. Utilizing data from the ENTSO-E Transparency Platform as a data source exemplar, it will compute the average electricity production from various types in a selected country. This will serve as a tangible demonstration of the platform's capabilities, illustrating how one can design an analytics platform on Azure Cloud. Moreover, the prototype is engineered with an emphasis on integrating Cloud and DevOps methodologies to ensure the robustness of continuous integration and deployment, thereby enhancing the platform's efficacy and adaptability for real-world usage.

1.1 ENTSO-E

The European Network of Transmission System Operators for Electricity (ENTSO-E) is an association that facilitates cooperation among European transmission system operators (TSOs). Comprising 39 TSOs from 35 countries, it oversees the operation of Europe's electricity grid, the world's largest interconnected electrical system. The organization aims to ensure a reliable electricity supply, support the transition to sustainable energy, and enhance the efficiency of the European internal electricity market, in line with mandates from EU legislation [1].

The ENTSO-E Transparency Platform aims to provide free, continuous access to pan-European electricity market data for all users, across six main categories: Load, Generation, Transmission, Balancing, Outages, and Congestion Management. This thesis focuses mainly on the Generation category, specifically on the "Actual Generation per Production Type" data view. This data view provides information on the actual generation of electricity per production type, showing real-time or historical generation data categorized by different types of energy sources or production technologies. Examples include Biomass, Fossil Gas, and Nuclear.

1.2 Topic Structure

The following aspects of the topic are going to be discussed thoroughly.

- **Background:** This chapter serves as the theoretical foundation, equipping readers with essential background knowledge necessary for understanding the project. It delves into foundational theories, key concepts, and relevant literature, establishing the academic context of the thesis project.
- **Requirements:** This chapter outlines the project's requirements in detail. It explores both user requirements – expectations from end-users – and functional requirements – specific behaviors and functions the project must exhibit.
- **Design:** The design process of the application is detailed in this chapter. It encompasses the architectural layout and the selection of technologies, including the rationale behind these choices. This section demonstrates how the design approach aligns with the previously outlined requirements, showcasing the decision-making process and its contribution to building an effective application.
- **Implementation:** This chapter narrates the transformation of designs and plans into a working application. It includes the development process, coding, integration of components, and innovative techniques employed. Challenges encountered and strategies used to address them are also key aspects. This section underscores the practical application of theoretical design, providing insight into the real-world functioning of the project.
- **Test and Evaluation:** The effectiveness of the application is assessed in this chapter. It outlines the testing methods used and includes a comprehensive analysis of test results. This chapter is pivotal in validating the project, providing an objective assessment of how well the application meets the outlined requirements.
- **Summary:** This chapter synthesizes all key points, recapitulating objectives, major findings, and overall significance. It reflects on the contributions made to the field and potential practical implications.
- **Bibliography:** This is the section where all the sources will be cited.

2. Background

2.1 Cloud

In the modern computing landscape, "cloud computing" involves utilizing servers accessible over the Internet for data storage and software execution. These servers are distributed across multiple geographical locations, negating the need for individuals or organizations to manage or own physical servers [2].

One of the significant advantages of cloud computing is its ability to facilitate data and software access from nearly any Internet-connected device. This convenience arises from the centralized storage and processing on these online servers, rather than on individual devices. For instance, should a laptop malfunction, important files stored on a cloud service such as Google Drive remain accessible from an alternate computer. The same principle applies to online email services like Outlook and additional data storage solutions, including Apple iCloud [2].

2.1.1 How cloud computing functions

Cloud computing operates using a technology known as virtualization. This technology creates a digital-only "virtual computer" that acts just like a real computer with its own hardware. Each of these digital computers, or "virtual machines," is separated from the others, even though they might share the same physical machine. This separation ensures that they don't interact with each other, keeping files and applications on one virtual machine hidden from the others [2].

Virtualization also lets one server act like many, which is good for making the most out of the hardware it's running on. In a way, a single data center can function like multiple data centers. This means cloud service providers can serve more customers at the same time without driving up costs. Even if one server in the cloud fails, the cloud service as a whole usually stays up and running. This is because cloud providers often use backup systems across different locations [2].

To use these cloud services, people connect via an app or a web browser over the Internet, regardless of the device they are using.

2.1.2 Cloud Infrastructure

Cloud infrastructure refers to the hardware and software components such as servers, storage, networking resources, and virtualization software that are needed to support the computing requirements of a cloud computing model. In essence, cloud infrastructure allows the management of

traditional workloads while also providing the flexibility to deliver new, cloud-native, and container-based applications [2].

2.1.3 Cloud Deployment Models

Cloud deployment models define a cloud environment's characteristics with respect to ownership, magnitude, accessibility, and purpose. These models pinpoint the server locations and the entities responsible for their management. Each model outlines the architecture of cloud infrastructure, detailing customization capabilities and whether services are pre-provided or self-constructed. Furthermore, these models establish the nature of interaction between the infrastructure and end-users. Below, various deployment models in the realm of cloud computing are delineated.

The three main different types of cloud infrastructure:

- **Public Cloud:** Public clouds deliver resources, such as compute, storage, network, develop-and-deploy environments, and applications over the internet. They are owned and run by third-party cloud service providers. The most known and used public cloud providers are AWS, Azure, and Google Cloud [3].
- **Private Cloud:** Private clouds are built, run, and used by a single organization, typically located on-premises. They provide greater control, customization, and data security but come with similar costs and resource limitations associated with traditional IT environments [3].
- **Hybrid Cloud:** Environments that mix at least one private computing environment with one or more public clouds are called hybrid clouds. They allow users to leverage the resources and services from different computing environments and choose which is the most optimal for the workloads [3].

2.1.4 Cloud services

The resources available in the cloud are known as "services," since they are actively managed by a cloud provider. A cloud service provider is an information technology company that provides on-demand, scalable computing resources like computing power, data storage, or applications over the internet. These services are sorted into several different categories, or *service models* [3].

For this project, the Azure cloud provider was selected based on the criteria outlined in the 'Requirements' chapter.

2.1.5 Azure Cloud Provider

Azure is a cloud computing platform offered by Microsoft that provides a wide range of cloud services, including compute, analytics, storage, and networking. Azure is a public cloud platform that offers more than 200 products and cloud services accessible over the public internet. Primary designed to help businesses manage challenges and meet their organizational goals by providing tools that support all industries, including e-commerce, finance, and a variety of Fortune 500 companies. Azure is compatible with open-source technologies, which gives users the flexibility to use their preferred tools and technologies [4].

2.1.6 Main service models of cloud computing

- **Software-as-a-Service (SaaS):** In this model, applications reside on cloud servers and are accessible through the Internet, eliminating the need for installation on individual devices. The arrangement can like renting a house: the landlord maintains the house, but the tenant mostly gets to use it as if they owned it. Notable examples of SaaS applications are Salesforce for managing customer interactions, MailChimp for mass emailing, and Slack for team collaboration [3].
- **Platform-as-a-Service (PaaS):** This model focuses on providing organizations with the resources needed to build their own applications, rather than hosting pre-built applications for them. PaaS suppliers provide a full set of development tools, infrastructure, and operating systems via the Internet. The situation is like renting all the tools and equipment necessary for building a house, instead of renting the house itself. Examples of PaaS providers include Heroku and Microsoft Azure [3].
- **Infrastructure-as-a-Service (IaaS):** This model allows organizations to lease servers and storage space from a cloud service provider. The leased infrastructure is then used by the organization to construct and manage their own applications. The arrangement is like a company leasing a plot of land on which they can build whatever they want — but they need to provide their own building equipment and materials. Notable IaaS providers include DigitalOcean, Google Compute Engine, and OpenStack [3].

Previously, the primary models of cloud computing were SaaS, PaaS, and IaaS, and nearly all cloud services could be categorized under one of these headings. Lately, however, a fourth model has come into play.

- **Function-as-a-Service (FaaS):** This model, often referred to as serverless computing, allows applications to be broken down into smaller, individual functions that operate only when activated. Think of it like a utility bill where one only pays for what is used, such as turning on a light only when needed and paying for just that usage [5].

Although the term is "serverless," these applications do indeed run on servers. The difference is that the servers are managed by the service provider, not the organization using the application.

Additionally, these serverless functions can automatically adjust to handle more or fewer users, like how a utility grid adjusts to supply power during peak or low demand. Serverless computing offers a flexible and cost-efficient approach to application development and management [5].

2.2 Infrastructure as Code

Infrastructure as Code (IaC) means using code to set up and manage infrastructure instead of doing it by manual processes. With IaC, simple config files are used to describe how things should be set up which ultimately makes changes and sharing easier.

Tools such as Terraform, AWS CloudFormation, and Ansible are widely used for this purpose. They provide the capability to set up the desired configuration in a file, which is then utilized to automate the setup process.

Advantages of Infrastructure as Code

- **Self-service:** When teams set up systems manually, often only a few individuals know the process and have the necessary access. This can lead to delays as the team expands. Using code for system setup allows for automation, enabling any developer to initiate the setup as needed [6].
- **Speed and safety:** Automating system setup through code is faster because computers operate more quickly than manual processes. It's also more reliable because computers follow instructions consistently, reducing errors [6].

- **Documentation:** Using code for system setup ensures that the system's workings are accessible to all. It acts as a guidebook that remains available even in the absence of key individuals [6].
- **Version control:** Storing different versions of setup code provides a record of all changes made. If issues arise, this record can be consulted to identify recent changes. If necessary, reverting to a previous version becomes an option [6].
- **Validation:** Defining system setup in code allows for thorough checks of every modification. The code can be reviewed, tested, and analyzed using tools, reducing the likelihood of errors [6].
- **Reuse:** Parts of the setup code can be repurposed for various projects. This approach avoids starting from scratch and builds on reliable, tested components [6].
- **Happiness:** Using code for system setup enhances job satisfaction. Manual, repetitive tasks can become tedious and stressful. Automation allows computers to handle repetitive tasks, while developers focus on creative aspects [6].

2.3 Virtual Machines

A virtual machine (VM) is an emulation of a physical computer, functioning as its digital counterpart. While the actual hardware is termed the 'host', the VM is often referred to as the 'guest'.

VMs enable the creation of multiple independent environments, each boasting its own operating system and applications, all on a singular physical machine. These VMs don't directly communicate with the host hardware. Instead, a software layer known as a 'hypervisor' acts as the mediator, allocating essential computing resources such as CPU, memory, and storage to each VM. This separation ensures VMs operate without affecting each other [7].

Virtualization allows a physical computer, often called a 'bare metal server', to separate its operating system (OS) and applications from its hardware. With the aid of a hypervisor, this computer can fragment itself into multiple "virtual machines" (VMs). Each VM operates its own OS and applications, yet shares the original server's resources, such as memory and storage.

The hypervisor acts as a manager, distributing the bare metal server's resources amongst these VMs while ensuring they operate harmoniously without interference [7].

There are two main hypervisor types:

- **Type 1 Hypervisors:** These run directly on the physical hardware, essentially replacing the OS. Software tools, like VMware's vSphere, help create VMs on these hypervisors, allowing the installation of guest OSs and the possibility of using one VM as a template for others [7].
- **Type 2 Hypervisors:** Operating as an application within an OS, these are commonly used for individual desktops or laptops. Users manually create a VM and decide on the allocation of physical resources. Advanced features might include options like 3D graphics acceleration [7].

2.4 Containerization

Containerization involves encapsulating software code with the necessary operating system (OS) libraries and dependencies into a single, streamlined unit termed a "container." These containers, lighter and more efficient than traditional virtual machines (VMs), serve as the foundational units for contemporary cloud-native applications [8].

The primary advantage of containerization is its ability to ensure software consistency across different environments. Traditional development often sees code behaving differently when shifted from one environment to another, leading to potential bugs. Containerization addresses this by bundling the application with all essential configuration files, libraries, and dependencies. This abstraction ensures that the container remains independent of the host OS, facilitating smooth execution across various platforms without compatibility issues [8].

Though the idea of containerization has been around for decades, its popularity surged with the introduction of the open-source Docker Engine in 2013. This provided the industry with a standardized approach to container creation, bolstering the widespread adoption of the technology. Nowadays, organizations increasingly lean on containerization for both developing new applications and updating existing ones for cloud compatibility [8].

Containers have the reputation of being "lightweight" because they utilize the host machine's OS kernel and don't need a separate operating system for each application. This compact nature allows them to occupy less space than a VM, reducing startup times and facilitating the concurrent operation

of multiple containers on the same computing resources. The result is better efficiency and reduced operational costs [8].

The most significant advantage of containerization is the ability to ensure that applications can be developed once and executed anywhere. This versatility accelerates the development process, safeguards against dependency on a single cloud provider, and offers benefits like simplified management, enhanced security, and fault isolation.

2.5 Virtualization vs. Containerization

As mentioned above, both containers and VMs allow multiple software applications to run in a single environment, optimizing compute efficiency by maximizing resource utilization and minimizing overhead. Yet, despite their similar objectives, there are clear and inherent differences in their design, functionality, and use cases that set them apart from one another.

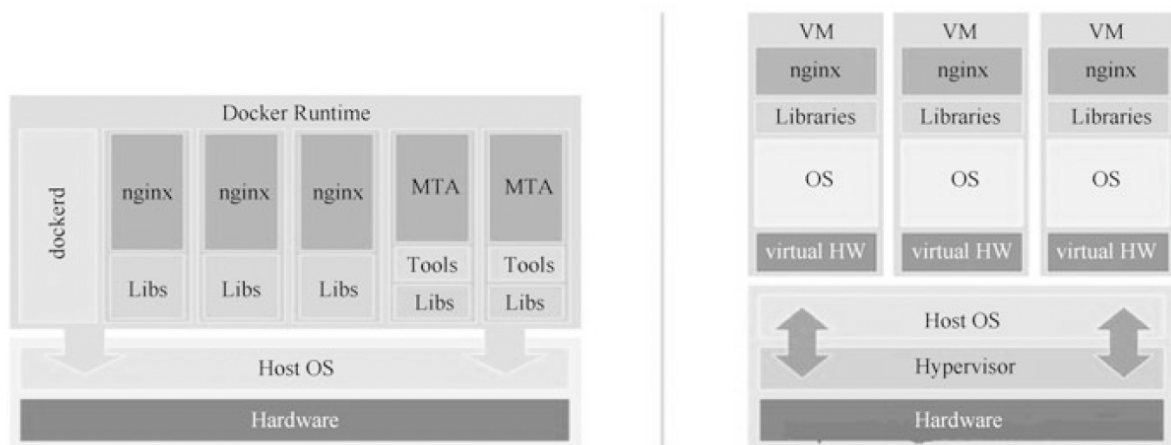


Fig 1. The difference between a container (left) and a virtual machine (right) [9]

Virtual Machines (VMs):

- VMs allow different operating systems and applications to run concurrently on a single computer. This means an organization could have both Linux and Windows or multiple OS versions on one server [10].

- Each application, alongside its associated files, libraries, dependencies, and even a copy of the OS, is packaged as a VM [10].
- By running multiple VMs on one machine, cost savings in capital, operations, and energy are optimized [10].

Containers:

- Containers further streamline resource use. They package application code with its associated configuration files, libraries, and dependencies into a singular executable software package [10].
- Unlike VMs, containers don't include a copy of the OS. All containers on a system share the host system's OS via the container runtime engine [10].
- Containers are "lightweight," sharing the machine's OS kernel. They don't have the overhead of an OS for each application, making them quicker to start and more resource-efficient than VMs. This leads to even greater savings in server and licensing costs [10].

2.6 Serverless

Serverless computing is an innovative paradigm in cloud services that promises to simplify the development and operation of applications by abstracting away the complexities of server management. Despite the term, serverless computing does not eliminate servers but makes their existence transparent to developers. It allows them to focus solely on writing code without worrying about the underlying infrastructure. This computing model operates on the principle of NoOps, meaning "no operations," where traditional server management and maintenance tasks are completely offloaded to the cloud provider [11].

NoOps approach is epitomized by Function-as-a-Service (FaaS) which is described on the beginning of this chapter under *Main service models of cloud computing* sub-Chapter, where individual functions are triggered by events or HTTP requests, executed in stateless containers that are spun up and down by the provider as needed.

The automated scaling or elasticity of serverless computing is one of its core features. It ensures that resources are dynamically allocated and billed based on actual usage, providing a fine-grained, pay-

as-you-go cost model. This means applications can scale from a few requests to millions without any manual intervention [11].

The evolution of serverless computing can be traced back to the advancements in virtualization and cloud services. It is the next step in the evolution of cloud offerings, following IaaS, PaaS, and SaaS, further reducing complexity and operational overhead for developers.

Serverless computing offers several advantages over traditional cloud service models. Here are some of the key benefits:

- **Cost-Efficiency:** With serverless, you only pay for the compute time you consume. There is no charge when your code is not running, making it an economical choice for various applications, especially those with variable workloads [11].
- **Scalability:** Serverless architectures inherently handle scaling automatically. As the number of requests for a function increases, the cloud provider automatically allocates more resources to handle the load [11].
- **Developer Productivity:** By abstracting away the server management aspect, developers can focus on writing code rather than worrying about the infrastructure. This leads to faster deployment of applications and features [11].
- **Quick Time-to-Market:** Serverless computing allows businesses to bring products to market more quickly due to the reduced operational complexity and the ability to rapidly deploy and update applications [11].
- **Improved Latency:** Serverless architectures can deploy instances of applications in multiple geographic regions, reducing latency by serving requests from the nearest data center [11].
- **Ecosystem and Community:** The growing serverless ecosystem provides a rich set of tools and services that can be integrated, offering solutions for monitoring, security, and continuous deployment, among others [11].
- **Focus on User Experience:** Freed from infrastructure concerns, organizations can dedicate more resources to user experience and innovation [11].

serverless computing represents a significant shift in how businesses deploy and manage applications, offering numerous benefits in terms of cost, scalability, and operational efficiency. As the market for serverless continues to grow, it is likely to become an even more integral part of the cloud computing landscape.

2.7 DevOps

DevOps is an updated software development approach that focuses on better coordination, communication, and automation between software developers and IT operations. This helps in delivering software more quickly and reliably. In the past, software developers and IT operations often worked separately, causing delays in software releases. DevOps aims to fix this issue by bringing these teams closer together. As a result, many companies are moving from old methods to DevOps [12].

Before DevOps, developers mainly worked on creating features, while the operations team looked after the stability and performance of these features. This separation sometimes led to problems such as system outages and misunderstandings. However, with DevOps—combining Development and Operations—there's a push for [12]:

- Making the work cycle faster.
- Quickly adding new features.
- Improving teamwork between development and operations.

By following this approach, teams work better together, communicate more, and use automation tools more often. This automation helps in reducing mistakes and makes processes like testing and deployment more efficient. [12]

2.7.1 DevOps Architecture

Both the development and operations teams are needed to release apps. The development side includes understanding needs, designing, building, and testing software. On the other side, operations manage these software components. With DevOps, the gap between these two sides is closed, which can lead to faster software delivery. The DevOps method is useful for various applications, like those on the cloud. It also helps in smoothly adding new features. When the two teams work separately, it takes more time to create and manage software [12].

Modern software development can be complex. Making decisions for such software can depend on many factors. DevOps helps complex software development handle this complexity by giving a way to always improve, try new things, and get feedback quickly. DevOps needs a structured setup to manage its tools and processes. A good setup can make software delivery efficient, accurate, and scalable. [12]

The DevOps lifecycle is a continuous software development process that employs DevOps best practices to plan, build, integrate, deploy, monitor, operate, and offer continuous feedback throughout the software's lifecycle. It is often represented by a continuous loop diagram as seen below –

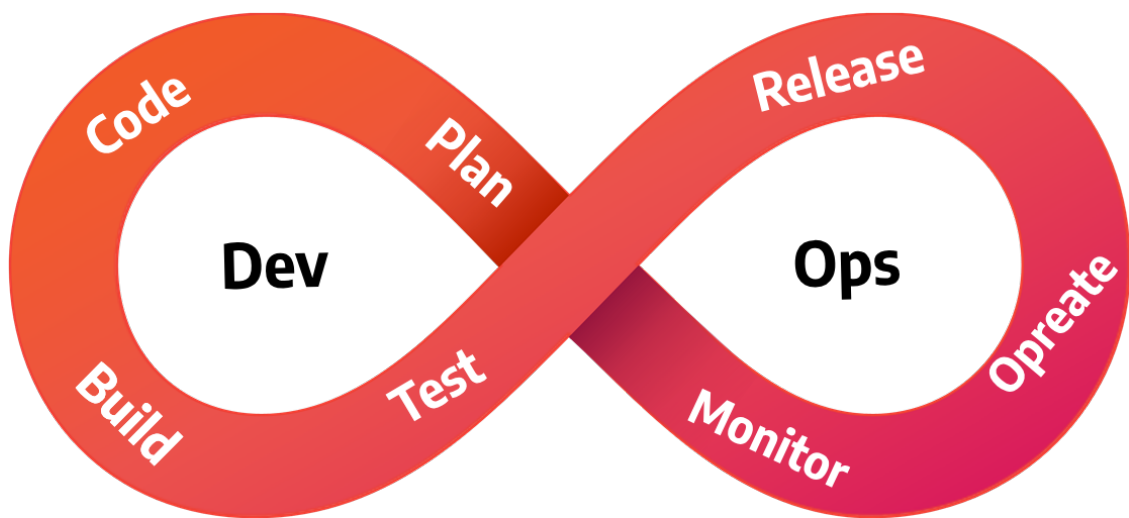


Fig 2. The DevOps lifecycle

- **Plan:** In the planning stage, teams map out the project. Unlike traditional ways of developing software, this method expects to go back and repeat steps when needed. This means we're always thinking about both past lessons and future cycles. All teams need to be involved to make sure nothing is missed [13].
- **Code:** In the coding phase, developers write the code, making it ready for the steps that come next. They follow the details from the planning stage, making sure the code fits the project's goals [13].
- **Build:** During this phase, new code is added to the project. If needed, the project is changed to fit this new code. Tools like GitHub are often used. Developers ask to add their code, it's

checked, and if it's good, it's added to the project. This way works well even when adding new things or fixing problems [13].

- **Test:** Here, the team checks everything to make sure the project works right. They also look for any special issues. An "edge case" is a problem that happens only in rare situations, and a "corner case" is when several unusual things happen at once [13].
- **Release:** This phase means the code is checked and ready to be used more widely. If everything is okay and there are no big problems, the project is ready to move on [13].
- **Deploy:** Now, the project is set up for everyone to use. While it used to be mostly the job of certain teams, in DevOps, everyone helps out. Working together like this makes sure everything runs smoothly [13].
- **Operate:** During this phase, the project is live, and people start using it. This isn't the last step. Instead, it helps guide what to do next, making sure everything works in the real world [13].
- **Monitor:** In this last stage, the team keeps an eye on how the project is used. They take note of feedback, issues, or ways to make it better. This information is used for the next cycle, helping to improve the whole process. [13]

3. Requirements

In this chapter, the MoSCoW prioritization technique [14] will be used to establish the criteria for goal setting. Only Three specific categories will be used to set the requirements: 'Must-haves' which are essential for completing the prototype, 'Should-haves' which are optional yet important initiatives that are not vital but add significant value and 'Could-haves' which are nice to have initiatives that will enhance the prototype if included. These requirements will be explained throughout the chapter and listed in full at the end.

3.1 Requirement Overview

The overall user requirement is to develop an interactive analytics platform prototype on Azure cloud. This platform should calculate the actual average electricity generated per production type for a specified country, using data from the ENTSO-E Transparency Platform as a proof of concept to show case how one cloud design an analytics platform on Azure cloud. The prototype should incorporate Cloud and DevOps practices to enable continuous integration and deployment.

The requirements for the prototype platform are divided into two main sections: "Functional Requirements" and "End user Requirements" These sections encompass the functionalities expected from the platform from a user's perspective and the underlying system operations necessary to facilitate those functionalities.

Functional Requirements

Functional requirements describe the operations and functionalities that the system needs to fulfill to support user functionalities and maintain smooth and secure operations:

- **Cloud Implementation:** The prototype platform should be implemented on Microsoft Azure cloud.
- **Infrastructure as Code:** The prototype platform should use infrastructure as code to create the necessary infrastructures. The chosen infrastructure as code tools and practices should be employed to automate the infrastructure setup.

- **Automated Pipelines:** The prototype platform should have automated pipelines for creating the infrastructure and deploying the application. The pipelines should ensure consistent and reliable deployment of the platform.
- **Infrastructure Destruction:** The prototype platform should provide a pipeline or mechanism to destroy the entire infrastructure when needed. The destruction process should be automated and ensure the removal of resources.
- **Partial Infrastructure Destruction:** The platform should allow for the destruction of stateless resources while preserving stateful resources.
- **Resource Split:** The prototype should be able to divide stateless and stateful resources.
- **Data Collection:** The data should be collected from ENTSO-E Transparency Platform via API request based on the user input.
- **Calculation:** As a proof of concept, the prototype platform should calculate the average value of the generated electricity per production type based on the input from the user using the data collected from ENTSO-E Transparency Platform.
- **Time Triggered Calculation:** The prototype platform should include a feature that triggers a scheduled job to calculate the monthly average value of electricity generated per production type for a specified country.
- **Data Storage:** The prototype platform should save both the user's input data and the computed average value in a type of database that is suitable for the data's nature and structure. This selected database should have the capacity to manage large quantities of data efficiently.

End User Requirements

User requirements detail the functionalities and features that will be directly accessed and used by the end-users of the platform:

- **Input Data:** The platform should feature a website equipped with a form for users to filter the required data. The form should prompt users to enter details such as the start date, end date, country, security token, and generation type so that the calculation can be performed using those inputs.
- **Data Presentation:** The prototype platform should display the stored data from the database back to the website. The data should include information such as the initial and end dates, country, generation type, and the calculated average value.
- **Data Deletion:** The platform should provide the ability to delete data directly from the website. Users should be able to select specific data entries for deletion.

Requirements Table

The requirements for judging the model can be grouped into two categories:

1. **Must-Have:** The success of the project hinges on these items. Their inclusion is non-negotiable, as the project would lack purpose without them. Simply put, it's a MoSCoW top-priority requirement.
2. **Should-Have:** These items are important but not as critical as the "must-have" ones. They are deemed a secondary priority, meaning they hold significance but aren't vital for success.
3. **Could-Have:** These items are desirable but not necessary. They rank below the first two categories, making them a third-tier priority. If their inclusion compromises cost or deadlines, they should be excluded. They should only be considered if they don't hinder other project aspects.

The following are all the requirements listed for the prototype' acceptance.

ID	Must/Should/Could Have	Requirement
1	Must	Cloud Implementation
2	Must	Infrastructure as code
3	Must	Automated Pipelines
4	Must	Infrastructure Destruction
5	Could	Partial Infrastructure Destruction
6	Could	Resource Split
7	Must	Data Collection
8	Must	Calculation
9	Could	Time Triggered Calculation
10	Must	Data Storage
11	Must	Input Data
12	Should	Data Presentation
13	Should	Data Deletion

Table 3.1: Requirements Table with each requirement having an ID number, whether the requirement is a must, could or should be had by the end of the development and implementation phase.

4. Design

This chapter focuses on the architecture design for an interactive analytics platform deployed on Azure cloud. As stated on the *requirement* chapter the platform aims to calculate the average electricity generation per production type for specific countries using open data from ENTOSE-E's platform. Two architectural approaches will be explored: a containerized Azure Web App Service and a solution utilizing Azure Functions.

The following sections will detail the technologies chosen for each architectural model and the reasons for their selection. The criteria for these choices include scalability, cost, performance, and deployment considerations based on the requirements given.

The architecture is designed with specific goals in mind, such as ensuring responsive performance, maintainability, and the capacity to handle varying workloads. The chapter will provide insights into how each architecture meets these objectives, presenting a clear view of the platform's design considerations.

4.1 Containerized Web App Service

A containerized Azure Web App is a cloud computing platform that enables the deployment and running of containerized web applications. Containers, often orchestrated with Docker, bundle an application with its environment and dependencies into a single package, providing consistency and isolation from the underlying infrastructure as stated in the *Background* chapter. This approach opens the possibility to build and deploy applications that can run anywhere, with the assurance that the behavior will be consistent across different environments.

For the interactive analytics platform, leveraging a containerized Azure Web App Service presents several practical benefits. The platform's deployment within Docker containers guarantees environmental consistency, ensuring that the future complex data processing behaves predictably across different deployment stages. This is crucial for maintaining the integrity of calculations and user experience. Additionally, the containerized approach allows for rapid deployment and iteration, a valuable feature for a prototype project where features and fixes need to be rolled out quickly. Azure's scaling capabilities are particularly advantageous for an analytics platform; they enable the service to adapt to varying workloads seamlessly, ensuring that performance remains stable even during peak data processing times. The integration with other Azure services simplifies the extension of the platform's capabilities, such as Cosmos DB and secure access management, without complicating the infrastructure. The managed environment of Azure App Service means less time spent on operational

overhead like maintenance and load balancing, freeing up resources to enhance the analytics functions. Moreover, the platform benefits from Azure's DevOps support which is one of the main requirements of the prototype project, facilitating a smooth continuous integration and deployment pipeline, which is essential for a platform that is expected to evolve rapidly. Finally, the pay-as-you-go pricing model of Azure App Service ensures cost-effectiveness, minimizing expenses during periods of low usage—a crucial factor for keeping the project economically viable while it scales according to demand.

4.1.1 Technologies and Tools

Version control system (Git)

In the realm of software engineering, collaborative projects often involve multiple developers making modifications to a shared code repository. To manage this collaborative effort effectively, there's a need for a mechanism that enables the team to make concurrent changes without overwriting each other's work and to maintain a record of the code's evolution.

Azure Repos is a set of version control tools that allow developers to manage their code and collaborate on code development. It provides two types of version control: Git, which is a distributed version control system, and Team Foundation Version Control (TFVC), which is a centralised version control system. Azure Repos includes free unlimited private Git repositories, making it easy to integrate it with Azure cloud. Hence this project aims to use Azure cloud as a cloud provider, Azure Repos has been chosen to be the version control system.

Terraform

Terraform offers a straightforward view of its functionality. It's an open-source tool developed by HashiCorp using the Go programming language. This Go code turns into a single tool named 'terraform' for each supported operating system.

This tool can set up systems from various devices, like a personal computer or a server, without needing additional setup. Internally, 'terraform' communicates with different service providers like Amazon Web Services (AWS), Azure, Google Cloud, DigitalOcean, OpenStack, and more using API calls. This means Terraform uses the existing infrastructure of these providers and the authentication methods already in place, such as existing API keys for AWS [6].

Terraform requires configurations to know which API calls to make. These configurations are text files that describe the desired system setup [6]. They represent the 'code' in 'infrastructure as code'.

Given the popularity of the software among Cloud Engineers and the features listed above Terraform has been chosen to be the IoC tool for provisioning the Azure cloud services.

Python Fast API

FastAPI is a modern, high-performance web framework for building APIs with Python 3.6+, leveraging standard Python type hints. Azure Functions, which will be discussed in more detail later, is a serverless compute service that allows for the execution of event-driven or HTTP-triggered code on a fully managed infrastructure [15].

This service supports frameworks compatible with HTTP-triggered Python functions, FastAPI included. Bearing this in mind, FastAPI is utilized to interact with ENTOSE-E's API to retrieve the necessary data for performing calculations, and then it stores the computed data into the database.

Azure Cosmos DB

Cosmos DB is a scalable, worldwide database from Microsoft Azure that's easy to manage and offers quick response times. It can handle various data models and query APIs. As a cloud-based NoSQL database, it's part of Azure's Platform as a Service (PaaS) [16]. Some even call it a serverless database due to its high availability, reliability, and data processing capacity.

The data produced by ENTOSE-E is NoSQL with an XML schema. To store this kind of unstructured data, Cosmos DB is ideal, as highlighted by its capabilities described in the previous paragraph.

Azure Key Vault

Azure Key Vault provides a centralized storage service for sensitive application data such as API tokens, app secrets, and passwords. This service eliminates the need to embed secret values within application code, allowing applications with the right permissions to retrieve secrets from the designated vault. [17, 16, 16]

To securely access data in Cosmos DB from backend, serverless services, or other Azure cloud resources, a connection string is required. This string provides connection details to an Azure Cosmos DB account, including account name, password, and endpoint. After creation, this connection string can be saved in Azure Key Vault. Access is then controlled by policy permissions, ensuring only applications or services with the correct permissions can retrieve data from Cosmos DB.

Azure DevOps

Azure DevOps, a SaaS offering from Microsoft, is tailored for managing DevOps environments, with a focus on agile software development and pipeline automation [18]. While it boasts a countless of features like artifacts, repos, pipelines, agile tools, and test plans, this project is going to only explore repos and pipelines.

A standout feature of Azure DevOps is the automation of pipelines, which are essential for the building, testing, and deploying of applications. The use of YAML for pipeline configuration is particularly advantageous because of its seamless integration with version control systems hosted on Azure repos, which ease future updates and changes. Due to these benefits, YAML configurations have been selected to automate creation of cloud resources and deployment processes within this thesis project.

Diving deeper into application configuration, environment variables are essential if one want to prioritize pipelines security and reusability. Within Azure Pipelines, these variables are organized into 'variable groups', which are accessible in the portal's Library section. These groups are vital for customizing the application during its journey through the pipelines. These variables can either be sourced from Azure Key Vault or manually entered in the portal.

The subsequent implementation chapter will delve into how these environment variables are utilized to store Azure credentials, terraform backend credentials, and frontend parameters, with a more in-depth discussion in the implementation segment.

Azure App Service and Container Registry

Azure App Service is a comprehensive, fully managed platform for creating, deploying, and expanding web applications. Its standout feature is its ability to autonomously oversee the underlying infrastructure, enabling developers to focus on crafting the code that powers their enterprises [19]. Azure App Service offers support for numerous programming languages, with Python being one of them, as well as developer tools and frameworks, encompassing both Microsoft-specific solutions and third-party software and systems.

The other feature of Azure App Service range of scalability options. As a Developer one can manually scale the app or let Azure automatically adjust the compute resources as demand for the web application changes or in case of this project when there is a need for more computation. It also integrates well with Azure DevOps and Azure Container Registry allowing for a continuous

deployment and version control. This means that developers can push updates and enhancements to their web apps in real-time, without worrying about downtime or manual deployment processes.

Azure Container Registry (ACR) is a private registry service that allows to build, store, and manage container images and artifacts for all types of containers deployment [20]. To deploy a containerized application to Azure App Service, the container image needs to be stored in a container registry such as ACR. Azure App Service can then be configured to pull the container image from the registry and run it as a web application [21]. For this project the custom created docker image with the necessary dependency requirements is going to be stored on ACR and run the application on Azure web app.

Azure Service Plan

As stated above Azure App Service is another platform-as-a-service (PaaS) offering from Microsoft Azure that creates the possibility to build, deploy, and scale web apps and APIs. An App Service plan is required to run an app service and it defines a set of compute resources for the app service to run on [22].

The cost of App Service plans is determined by the plan's tier and the quantity of instances. These tiers encompass Free, Shared, Basic, Standard, Premium, and Isolated [23].

The Free and Shared tiers are primarily for development and testing which is being used for this prototype project, whereas the Basic, Standard, and Premium tiers are designed for production workloads and operate on dedicated virtual machine instances.

4.1.2 Architecture Design for Containerized Web App

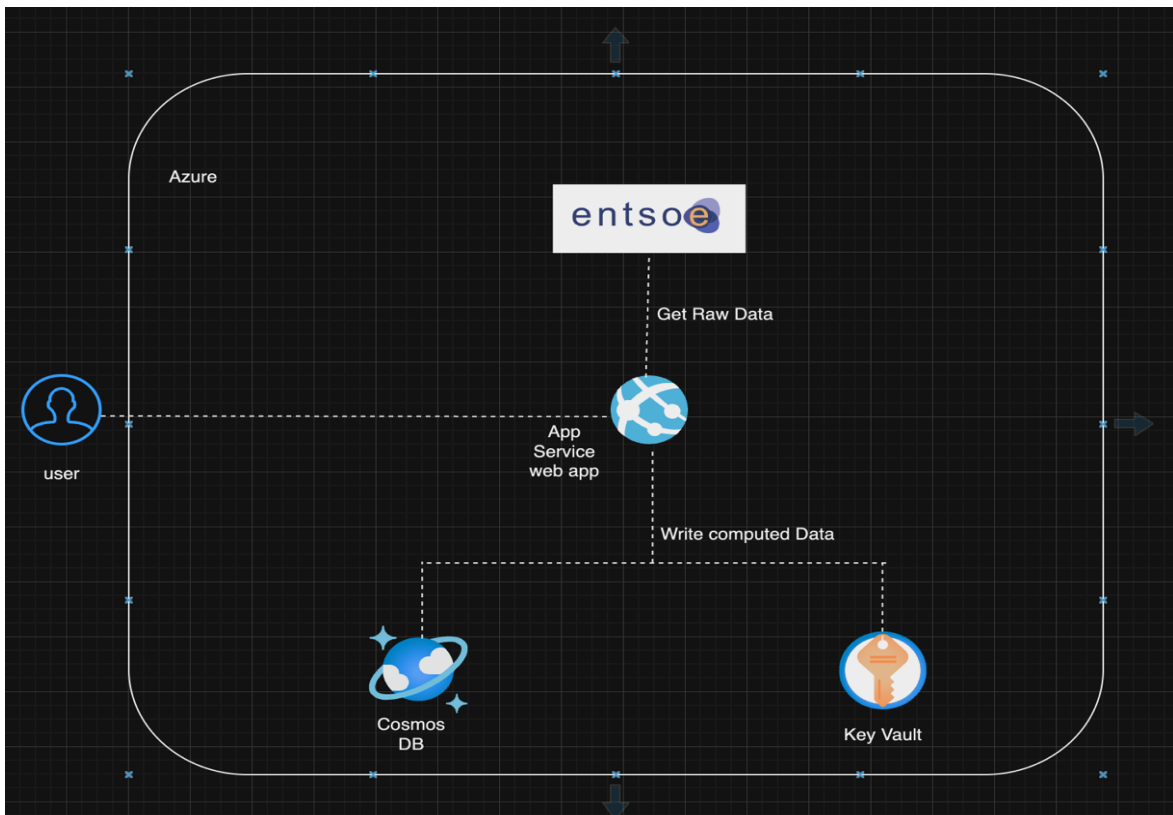


Fig 3. Architecture design for containerized web app

The Architecture consists of an Azure App Service Web App designed to fetch and process data from the ENTOSE-E API and subsequently store the computed average results in a Cosmos DB..

- **Functionality:**
 - Receives specific data parameters from a user via an API call.
 - Makes an API request to ENTOSE-E using the provided data to retrieve raw data.
 - Calculates the average value based on the retrieved data.
 - Stores the computed average value in Cosmos DB.
- **Security:** Accesses Cosmos DB using a connection string securely stored in an Azure Key Vault.

Azure Web App Service is primarily designed for hosting web applications, RESTful APIs, and backend services. It offers a robust environment for continuous operation and meets the "Must-Have" requirements outlined in the *Requirements table* from the second chapter. Azure Web App is best suited for applications with continuous traffic or those requiring persistent server connections. If the prototype expects constant user traffic, this architectural design is the most suitable choice in this scenario. However, it's worth noting that Azure Web App doesn't inherently support event-driven execution like Azure Functions does, which is going to be explored in detail next.

4.2 Azure Functions App

Azure Functions operates as a serverless computing platform, enabling developers to establish event-driven applications without the complexities of infrastructure management. This platform accommodates various programming languages, such as C#, JavaScript, F#, Java, and Python. Azure Functions presents a range of hosting alternatives tailored for diverse business and application demands, ranging from entirely serverless configurations with billing based on execution time, to traditional hosting approaches like Azure App Service plans and Premium plans [24].

Additionally, Azure Functions can activate processes through different triggers, like HTTP endpoint access or a predetermined timer, and integrates seamlessly with resources, including storage and queues, through specialized bindings [25].

In the *Requirements* chapter, it is specified that the system could support two distinct calculation processes: one that responds to data input by the user, and another that operates on a predetermined schedule. Azure Functions provides the necessary capabilities to meet these requirements. It allows for the implementation of an HTTP-triggered function to handle on-demand calculations when data is supplied by the user. Additionally, Azure Functions offers a timer-trigger feature to automatically carry out calculations at scheduled intervals.

4.2.1 Architecture Design for Azure functions

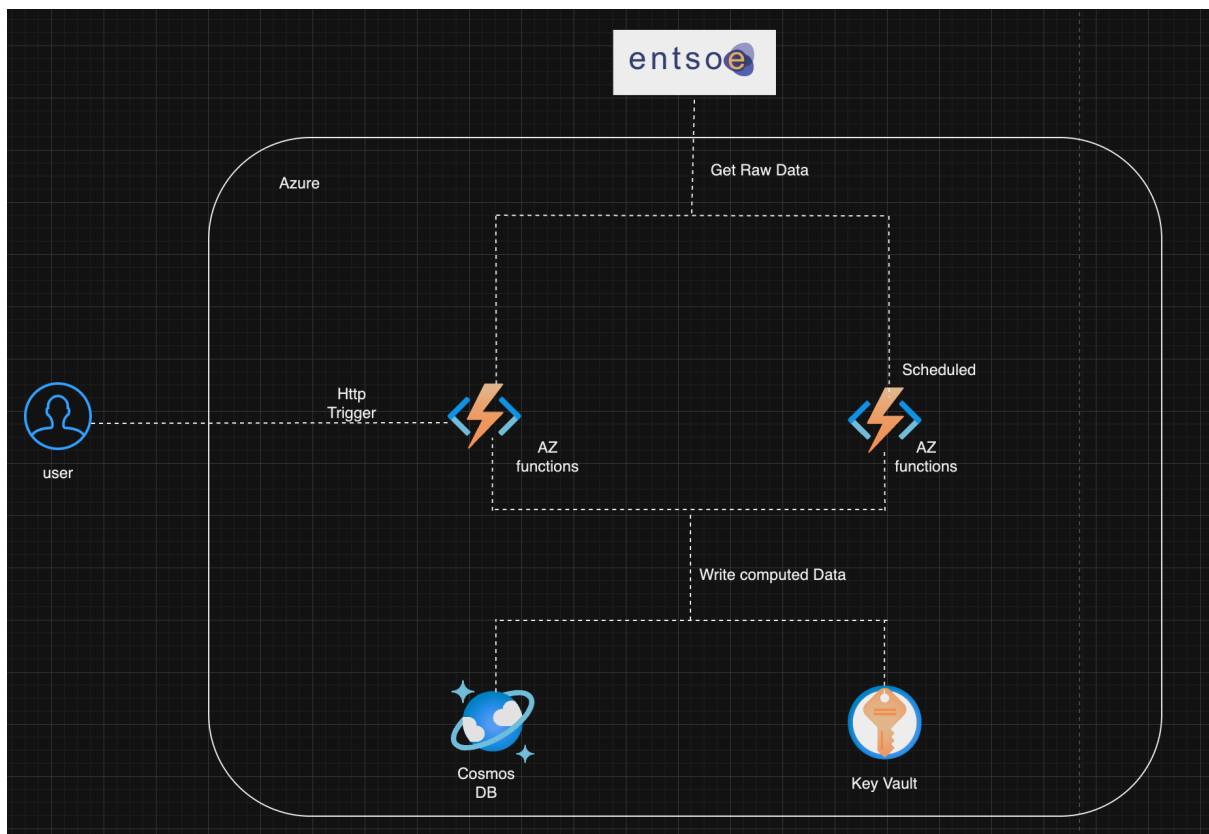


Fig 3. Architecture design for azure functions

This time the above Architecture consists of two Azure Functions designed to fetch and process data from the ENTPOSE-E API using two event driven triggers and subsequently store the computed average results in a Cosmos DB.

1. HTTP API Triggered Azure Function:

- **Trigger:** Activated by an HTTP API call.
- **Functionality:**
 - Receives specific data parameters from the frontend provided by the user.
 - Makes an API request to ENTPOSE-E using the provided data to retrieve raw data.
 - Calculates the average value based on the retrieved data.
 - Stores the computed average value in Cosmos DB.
- **Security:** Accesses Cosmos DB using a connection string stored securely in an Azure Key Vault.

2. Scheduled Time Triggered Azure Function:

- **Trigger:** Activated automatically on the 3rd day of every month.
- **Functionality:**
 - Fetches data from the ENTOSE-E API based on predefined filtered values.
 - Calculates the monthly average value from the fetched data.
 - Stores the computed monthly average value in Cosmos DB.
- **Security:** Uses the same secure connection string from the Azure Key Vault to access Cosmos DB.

Azure Functions offer a key advantage for applications that don't run continuously but need to respond to specific events, such as a new data entry from the user or the monthly scheduled job. This contrasts with Azure Web App Service, which is designed to run continuously and handle regular web traffic.

For a prototype that processes information occasionally say only a few times a day or week and serves a small number of users, Azure Functions is a suitable choice. It's activated only when needed, runs the necessary code, and then shuts down, which can keep costs low since billing is based on the number of times the function is triggered and the duration of its execution.

However, this pay-per-use model could lead to unpredictable costs. If the application suddenly gets more traffic and the functions are triggered more often than expected, the cost could increase quickly. This means that while Azure Functions can be cost-effective for infrequent tasks, there's a risk of cost spikes if usage patterns change and become less predictable.

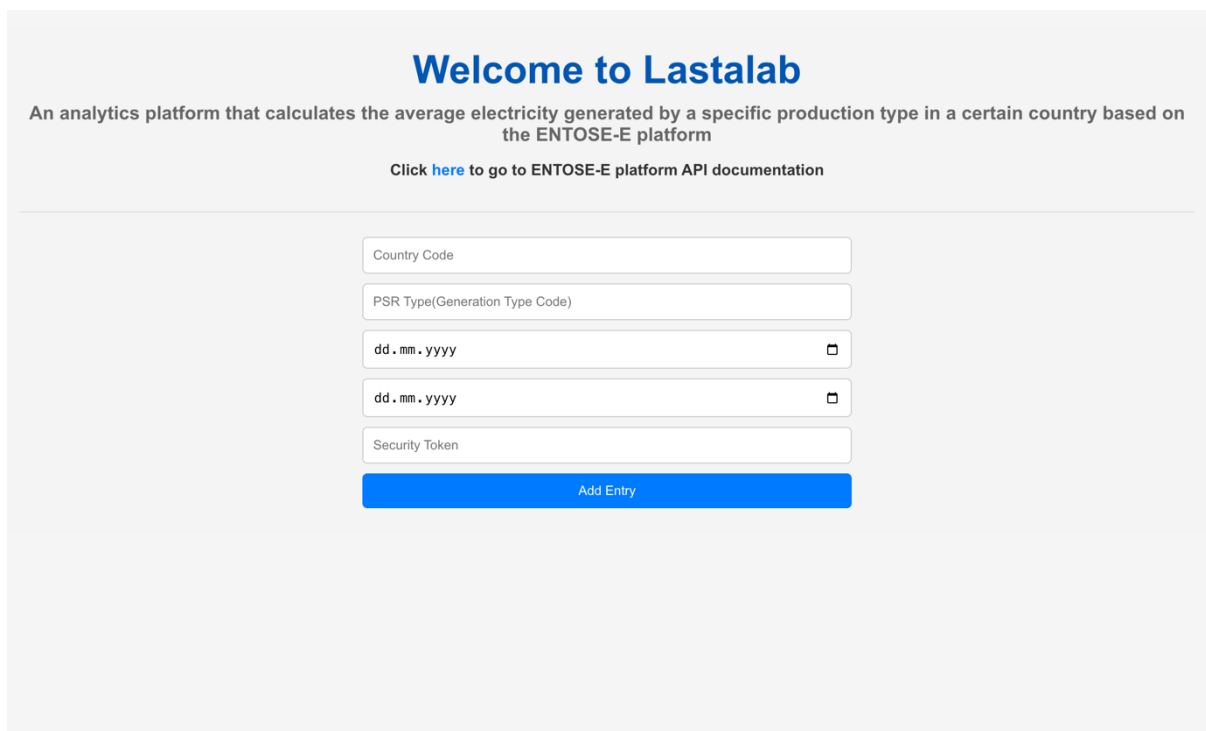
When planning to use Azure Functions, it's important to estimate how often the functions will run to understand potential costs better. The service is great for saving money when usage is low, but it's also crucial to monitor it to avoid unexpected charges if the frequency of the functions' triggers increases.

4.3 Frontend

Azure Static Web Apps is a service that simplifies the hosting and deployment of static web content, serving HTML, CSS, JavaScript, and image files directly from a content delivery network (CDN). It's optimized for static websites and web applications that rely on client-side processing. [26]

React is a popular open-source JavaScript library for building user interfaces, particularly for single-page applications where you need a responsive and dynamic client-side user experience. React's component-based architecture makes it a good fit for modern web applications, allowing developers to create large web applications that can change data, without reloading the page [27].

Azure Static Web Apps service is utilized to host the front end of the project, which is developed using React. This setup provides a robust platform for delivering static content, augmented by React's capabilities for creating dynamic and responsive user interfaces.



The screenshot shows a web application interface with the following elements:

- Title:** "Welcome to Lastalab" in blue.
- Subtitle:** "An analytics platform that calculates the average electricity generated by a specific production type in a certain country based on the ENTOSE-E platform".
- Link:** "Click [here](#) to go to ENTOSE-E platform API documentation".
- Form Fields:**
 - Country Code
 - PSR Type(Generation Type Code)
 - Start Date: dd . mm . yyyy
 - End Date: dd . mm . yyyy
 - Security Token
- Submit Button:** "Add Entry" (blue background).

Fig 5. *Frontend application design*

The front-end application features a form that allows users to input several parameters: the country, production type, start date, end date, and a security token obtained from the ENTOSE-E website. This form is designed to capture user input necessary for querying the analytics platform. A submit button is provided to add the entry to the system.

Additionally, the interface includes a section where users can view the calculated average values retrieved from the database. This data is presented along with associated start and end dates, as well as a unique identifier for each entry. To facilitate data management, each data point is accompanied by a delete button, enabling users to remove entries from the database as needed.

This interactive and user-friendly interface ensures that users can easily interact with the analytics platform, inputting and managing their data with ease while utilizing the security measures provided by the ENTOSE-E token system.

5. Implementation

This proof-of-concept project has been developed on a MacBook Pro utilizing macOS 14 Sonoma. All the utilized command-line tools and commands have been executed in the macOS Terminal. It's important to note that while this document doesn't include specific instructions for Windows and Linux, equivalent tools and commands do exist for those operating systems. For code editing and database management, Visual Studio Code and Azure Data Studio were used, respectively. Both tools are available and compatible with Windows and Linux platforms.

5.1 Initial Configuration and Local Environment Setup

The implementation of our infrastructure as code (IaC) begins with the setup of a local development environment that facilitates the deployment of resources to the Azure cloud platform. The configuration tasks involve the following key steps:

1. **Install Terraform:** Terraform is an open-source IaC tool that enables the definition and provisioning of cloud infrastructure using a high-level configuration language known as HashiCorp Configuration Language (HCL). To begin, Terraform must be installed on the local machine.
2. **Set Up Azure CLI:** The Azure Command-Line Interface (CLI) is a set of commands used to manage Azure resources. It is necessary to install the Azure CLI to enable Terraform to interact with Azure.
3. **Create Azure Service Principal:** Terraform requires authentication to manage resources within Azure. This authentication is provided through an Azure Service Principal, which is an identity created for use with applications, hosted services, and automated tools to access Azure resources. This can be achieved via the following Azure CLI command:

```
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/YOUR_SUBSCRIPTION_ID"
```

This command will output information that includes an application ID, password, and tenant ID, which are used to configure Terraform's Azure provider and later this information is also used to set up the automatic pipelines.

5.2 Infrastructure

Before diving into the implementation details, it's important to understand the concept of Terraform modules while dealing with infrastructure. Modules in Terraform are self-contained packages of Terraform configurations that are managed as a group. Modules are used to create reusable components, improve organization, and to treat pieces of infrastructure as a black box. [28]

5.2.1 Code Structure and Best Practices for Terraform

Incorporating to Terraform best practices is essential for maintainability and reusability. Well-structured and modular code is easier to maintain, understand, and scale, which is particularly beneficial as infrastructure complexity increases. This structured approach also facilitates collaboration, as it enhances readability and allows for more straightforward navigation of the codebase. Best practices also improve version control processes by logically organizing the code into manageable units, reducing the risk of conflicts and errors [29].

- **main.tf:** file contains the definitions of the resources that Terraform will manage. It is where the modules get invoked, for instance, to create resources like Cosmos DB and Key Vault.
- **variables.tf:** Here, variables are declared that will be consumed within the Terraform configurations. This allows for a dynamic setup that can be customized for different environments or deployments.
- **outputs.tf:** Outputs are like return values for a Terraform module. This file declares the output values that we can use to display or use elsewhere in our Terraform configuration.
- **providers.tf:** In this file, we define the providers that Terraform will use to interact with the cloud services. For Azure, we will configure the Azure provider with the credentials obtained from the service principal.
- **terraform.tfvars:** Here, the actual values are specified for the variables defined in **variables.tf**. The **terraform.tfvars** file are automatically loaded by Terraform to populate variables, also making it an ideal place to store environment-specific configurations for the future.

5.2.2 Database

The Terraform configuration file which is found under *infra/database* directory of the project repository sets up an Azure environment with two main components: an Azure Cosmos DB account and an Azure Key Vault.

- **Resource Group:** It starts by creating a Resource Group called "**rg-stateful-lastalab**", which is a logical container for grouping related stateful Azure resources for the project.
- **Cosmos DB:** It then uses the module *cosmosdb* which is found under *modules* directory of the project to deploy Azure Cosmos DB, a globally distributed, multi-model database service. The module is parameterized to customize the deployment based on the project's needs and can easily be modified for future by changing the parameters from *terraform.tfvars* file.
- **Key Vault:** Following that, another *keyvault* module is used from the same *modules* directory to deploy Azure Key Vault, a service that safeguards cryptographic keys and other secrets used by cloud applications and services. The Key Vault is configured to store the connection string of the Cosmos DB as a secret, which enhances the security of sensitive data.

All the terraform configuration files are designed to be modular and reusable, and it incorporates variables to allow for flexibility and customization of the Azure resources according to the project's requirements. It also includes tagging of resources for better management and identification within Azure. After creating the Cosmos DB, the next step is to create backend infrastructure for the containerized service web app.

5.2.3 Web App Backend

The Terraform configuration file, located in the *infra/backend* directory of the project repository, orchestrates the setup of an Azure environment customized for a stateless web application. Before creating the stateless web application, a resource group named "**rg-stateless-lastalab**", is established in the *infra/stateless-rg* directory to group all the stateless resources in one location. This also facilitates the simultaneous deletion of the stateless resources, fulfilling one of the 'Cloud Have' requirements from the second chapter.

- **Azure Data Retrieval:** The main.tf file starts by gathering information about the current Azure client configuration, as well as details of existing Azure Resource Groups and a Cosmos DB account.

- **Azure Key Vault:** Reads data from an Azure Key Vault to obtain the Cosmos DB connection string, which is crucial for the web application's database operations.
- **Azure Container Registry (ACR):** Provisions an ACR for storing Docker images, using the name derived from the project name and located in the stateless resource group.
- **Service Plan & Web App:** Establishes a Linux-based Azure Service Plan and Web App in the stateless resource group. The Web App is configured with environment variables including the Cosmos DB connection string from the Key Vault.
- **Managed Identity & Role Assignment:** Sets up a system-assigned managed identity for the Web App and assigns it a role to pull images from the ACR. This identity is also granted permissions to access secrets in the Key Vault.
- **Key Vault Access Policy:** Updates the Key Vault access policy to allow the Web App's managed identity to perform operations on secrets within the Key Vault.

The provided **terraform.tfvars** file supplies the necessary values for variables like project names, resource group names, and the Key Vault name, ensuring the resources are correctly labeled and associated with the project. This script ensures that the web application has secure access to the database connection string and manages container deployment effectively. While it's possible to skip the creation of the Azure Function for now, as it's part of the second infrastructure design, it's better to proceed with it at this stage since it relates to Terraform, which is currently under discussion.

5.2.4 Serverless Azure Functions

The Terraform configuration file, situated within the *infra/serverless* directory of the project's repository, creates a serverless application infrastructure in Azure. Here's what it accomplishes:

- **Azure Context Retrieval:** It starts by collecting information about the current Azure client and the details of existing Resource Groups, along with the Cosmos DB account data same as the backend configuration.
- **Key Vault Secrets:** Accesses an Azure Key Vault to retrieve the Cosmos DB connection string, ensuring secure database communication for the serverless application.

- **Storage Account:** Provisions an Azure Storage Account in the stateless resource group, which is essential for storing data and files used by Azure functions.
- **Service Plan:** Creates a service plan under the name provided by "`lastalab-app-service-plan`", which is a prerequisite for hosting serverless functions on Azure.
- **Application Insights:** Sets up Azure Application Insights with the name `lastalab-app-insights` for monitoring and analyzing the performance and usage of the serverless application which is going to be useful for future.
- **Linux Function App:** Deploys an Azure Linux Function App configured for Python runtime. It uses the created storage account and service plan and is tailored with CORS settings and application insights. The function app is also configured with environment variables that include database connection details fetched from the Key Vault.

5.2.5 Static Web App Frontend

The Terraform configuration located in *infra/frontend* directory of the project repository, start with the similar approach like web app backend and serverless function app configuration, it retrieves necessary Azure client and Resource Group context and then proceed to provisioning the following resource to create a secure, scalable, and easily managed environment for serving a frontend static web application, with sensitive information such as deployment tokens securely handled by Azure Key Vault.

- **Static Site:** Provisions an Azure Static Site for hosting static content. It's a scalable and serverless hosting service for static web applications, configured under the stateless resource group with the provided project name.
- **Key Vault Secret:** Updates the Azure Key Vault with a new secret holding the Azure Static Site's API key. This allows for secure storage and retrieval of the API key, which is essential for automated deployment processes to the static site.

The reason for having a static web app is to enable the development of an independent frontend that can function effectively for both the web app and serverless components.

5.3 Source Code

In this subchapter, the focus will be on the creation of the backend API integration with the ENTOSE-E platform. It will also delve into **how this computed value is sent to Cosmos DB for storage**. This process will be explored for both the containerized web app and serverless Azure Functions.

5.3.1 FastAPI Web Service

The source code file located in the *src/backend* directory of the project creates the necessary backend API integration to interact with the ENTOSE-E platform API, fetching filtered data based on user input. It begins the process,

- **CORS Middleware:** It configures Cross-Origin Resource Sharing (CORS) to allow requests from any origin, enabling the web service to be called from different domains.
- **Data Retrieval Endpoint (*/get_all_data/*):** This endpoint fetches all the data stored from the Cosmos database, assigns a string ID to each data, and returns them so that the data can be displayed on the frontend .
- **Data Filtering and Insertion Endpoint (*/get_filtered_data/*):** This endpoint accepts a new entry from the user, filters out the security token for security purposes, and performs the average calculation on the filtered data from ENTOSE-E API response then stores it in the Cosmos database.
- **Data Deletion Endpoint (*/delete_data/{id}*):** This endpoint deletes an entry from the database using its ID. If the entry is not found, it raises an HTTP 404 error.
- **Data Models:** EntryIn is for input data from the user, which includes Country, PSR Type (Electric Generation type), Security Token, and start and end dates. EntryDB is for Cosmos database entries.
- **Dockerfile:** The docker file located located in *src/backend* helps to set up a Docker container with Python 3.10, installs dependencies from a requirements.txt file, and runs the FastAPI application using Uvicorn.
- **Configuration settings:** to manage environment variables that configure the connection to the Cosmos DB, such as endpoint, key, database name, and collection name.

Overall, this service provides a RESTful API to interact with a Cosmos Database and integrates with an external energy data API, demonstrating data ingestion, processing, and CRUD operations within a Dockerized Python application.

5.3.2 Serverless function app

The FST API python script *function_app*, located in the *src/serverless* directory, is designed to work with Azure Functions, a serverless compute service that executes event-triggered code without the need for explicit provisioning or infrastructure management. The script defines a timer-triggered function that calculates the monthly average electricity generated from Biomass in Germany every 3rd day of the month. It utilizes the same API integration, data modeling, and data processing as the Fast API Web service, enabling computation based on user input as well. This Azure Function is intended for deployment in the cloud, where it operates in a fully managed environment. It's scheduled to execute without manual intervention if needed.

5.4 Pipelines

The *azdo* directory of the project serves as the foundation for defining DevOps processes within the project. It is organized into subdirectories that cater to different aspects of infrastructure management and application deployment.

azdo/infra: Within the **azdo/infra** directory, there are YAML files responsible for managing the lifecycle of infrastructure on Azure through Terraform. These pipelines are manually triggered, which allows for controlled execution of infrastructure changes. The pipelines include stages for planning, which generates a Terraform execution plan, and applying, where the plan is executed to provision, update and destroy the infrastructure. This setup is crafted to manage a variety of resource types in the Azure cloud environment.

azdo/code: The YAML files in the **azdo/code** directory define pipelines for continuous integration and deployment of applications. These pipelines are configured to respond to changes in the main branch of the repository, building and deploying the appropriate application components:

- For the backend, the pipeline builds a Docker image and pushes it to the Azure Container Registry. Subsequently, it deploys the image to an Azure Web App.
- The serverless application deployment is automated to Azure Functions directly upon updates.
- The frontend deployment pipeline is designed to deploy static content to Azure Static Web Apps.

Each YAML configuration is tailored to the specific deployment needs of the application component it serves, ensuring that the build and deployment processes are optimized for backend, frontend, and serverless architectures.

azdo/scripts: Scripts located in the **azdo/scripts** directory are utilized to configure and initialize the Terraform backend, crucial for state management in Terraform. These scripts facilitate the setup of a secure Azure Storage Account for storing the Terraform state, enabling consistent infrastructure management practices.

Integration into Azure DevOps

The YAML files are integrated into Azure DevOps by creating pipelines that reference these files. This integration allows for the automation of build and deployment processes, as well as infrastructure provisioning within Azure DevOps, leveraging the version-controlled configurations for consistency and repeatability.

This structure supports the DevOps goal of automating and streamlining the build, deployment, and infrastructure management processes to facilitate a smooth and efficient continuous integration and deployment workflow.

6. Test and Evaluation

This chapter presents a comprehensive evaluation of the interactive Azure cloud-based analytics platform designed for this project. The platform's core function is to enable users to filter data from the ENTOSE-E platform based on specific parameters, facilitating the calculation of average electricity generation values for different generation types in various countries. The evaluations focus on the platform's pipeline operations, functional capabilities, and scheduled computational tasks. Emphasis is placed on ensuring that the platform not only meets the specified requirements but also provides reliability and efficiency in its operations.

Test Case 1: Pipeline Resource Management

The first test case involves the examination of the pipeline's ability to manage resources effectively. This process includes the creation, deployment, and destruction of resources on Azure Cloud through Azure DevOps. The test scenario involved executing respective pipelines and observing their performance in handling these tasks.

Results: The pipelines demonstrated a high level of efficiency and accuracy. They successfully created and deployed necessary resources to build the prototype platform, and also managed the partial destruction of stateless resources. This test case confirmed the reliability and effectiveness of the pipeline in resource management.

Requirement list for the above test cases:

Requirements	Done
Cloud Implementation	Yes
Infrastructure as code	Yes
Automated Pipelines	Yes
Infrastructure Destruction	Yes
Partial Infrastructure Destruction	Yes
Resource Split	Yes

Test Case 2: Functional Capability of the Prototype

The second test case focused on the platform's functional capability, specifically its ability to process user inputs, perform calculations, display results and data deletion .

The test used a sample input:

Country: 10Y1001A1001A83F

PSR Type: B01

Start Date: 01/02/2023

End Date: 07/02/2023

Security Token: xxx-xxx

Results: The platform successfully utilized these parameters to send an API request to the ENTOSE-E platform. It then accurately calculated the average value based on the provided parameters. Furthermore, the platform effectively stored the user parameters, a unique identifier, and the calculated average value in the database. Users were able to view this stored data, complete with a deletion option, on the web application. This test case affirmed the prototype's ability to handle functional requirements efficiently.

Requirement list for the above taste cases:

Requirements	Done
Data Input	Yes
Data Collection	Yes
Calculation	Yes
Data Storage	Yes
Data Presentation	Yes
Data Deletion	Yes

Test Case 3: Scheduled Monthly Average Computation

The third test case assessed the Azure Function serverless design resource's capability to perform scheduled monthly average computations.

Results: This function demonstrated consistent performance while being tested using Azure functions test cases, furthermore it successfully completed the scheduled tasks. Monthly computations were performed accurately, showcasing the platform's capability to handle automated, time-bound tasks effectively.

Requirement list for the above taste cases:

Requirement	Done
Time Triggered Calculation	Yes

7. Summary

This thesis project achieved its goal by designing and developing a prototype for an analytics platform hosted on Azure cloud. This platform provides users with insightful analytics about electricity generation. It uses data from the ENTSO-E Transparency Platform to calculate the average electricity produced by different types of production in various countries, demonstrating its practical use.

A key aspect of the project's success was the integration of Cloud and DevOps methodologies. This integration enabled the implementation of continuous integration and deployment processes, improving the platform's efficiency and adaptability for real-world applications. The developed pipelines are essential, allowing for the creation, partial distribution, and full destruction of resources, all managed efficiently through the pipelines.

The thesis also aimed to explore two distinct prototype architectural approaches to design interactive analytical platform. The first, a Containerized Azure Web App Service, is ideal for applications that expect continuous traffic but has limitations in supporting event-driven execution. The second, Azure Functions, is designed for on-demand calculations and is better suited for event-driven, on-demand computational tasks. This exploration provides insights into the strengths and applications of each architecture design, suggesting the Azure Web App Service for scenarios with constant traffic and Azure Functions for event-driven data processing.

The prototype's design is extensible, which means it can adapt to future technological advancements and changing requirements. This dual architectural approach helps in evaluating and selecting suitable technology as the needs of the platform evolve, ensuring its long-term relevance and scalability. This project can also be extended by combining two of the designs when there is a need for both continuous traffic and event-driven data processes. In conclusion, the thesis not only delivers a functional analytics platform prototype but also contributes to the understanding of Cloud and DevOps integration in real-world applications. It highlights the potential of cloud-based solutions in data analytics and emphasizes the importance of adaptable architecture in the rapidly evolving technology landscape.

Bibliography

- [1] entose, "ENTSO-E Mission Statement," [Online]. Available: <https://www.entsoe.eu/about/inside-entsoe/objectives/>. [Accessed November 2023].
- [2] B. Okhuoya and B. Uzoma, "Cloud computing," *A research on cloud computing*, 2022.
- [3] A. Ishart, "Cloud Computing - A Comprehensive Definiton," *Journal of Computing and Management Studies*, 2017.
- [4] Microsoft Azure, "Azure Cloud," [Online]. Available: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure>. [Accessed October 2023].
- [5] cloudflare, "what is the cloud," [Online]. Available: <https://www.cloudflare.com/en-gb/learning/cloud/what-is-the-cloud/>. [Accessed October 2023].
- [6] Y. Brikman, *Terraform up & running*, O'Reilly, 2017, pp. 19-22.
- [7] IBM, "What are virtual machines," 2023. [Online]. Available: <https://www.ibm.com/topics/virtual-machines>. [Accessed October 2023].
- [8] IBM, "What is containerization," 2023. [Online]. Available: <https://www.ibm.com/topics/containerization>. [Accessed 10 2023].
- [9] Huawei Technologies, *Cloud Computing Technology*, Hangzhou, Zhejiang, China: Posts & Telecom Press, 2021.
- [10] IBM, "Containers vs. Virtual Machines," [Online]. Available: <https://www.ibm.com/blog/containers-vs-vm/>. [Accessed October 2023].
- [11] N. H. Samuel Kounev, "Serverless Computing: what it is, and what it is not?," 2023.
- [12] R. T. Amitkumar V. Jha, "From theory to practice: Understanding DevOps culture and mindset," *Cogent Engineering*, pp. 7-9, 2023.
- [13] S. Das, "DevOps Lifecycle : Different Phases in DevOps," 24 02 2023. [Online]. Available: <https://www.browserstack.com/guide/devops-lifecycle>. [Accessed 10 2023].
- [14] A. Business, "MoSCoW Prioritisation," [Online]. Available: <https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html>. [Accessed October 2023].
- [15] Microsoft Azure, "Using FastAPI Framework with Azure Functions," June 2023. [Online]. Available: <https://learn.microsoft.com/en-us/samples/azure-samples/fastapi-on-azure-functions/azure-functions-python-create-fastapi-app/>. [Accessed October 2023].
- [16] Microsoft Azure, "Azure cosmos DB," February 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/cosmos-db/resource-model>. [Accessed October 2023].
- [17] Microsoft Azure, "About Azure Key Vault," March 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/key-vault/general/overview>. [Accessed October 2023].
- [18] Microsoft Azure, "Azure DevOps," [Online]. Available: <https://azure.microsoft.com/en-us/products/devops#features>. [Accessed October 2023].

- [19] Microsoft Azure , "App Service," [Online]. Available: <https://learn.microsoft.com/en-us/azure/app-service/overview>.
- [20] Microsoft Azure, "Introduction to Container registries in Azure," June 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-intro>. [Accessed October 2023].
- [21] Microsoft Azure, "Continuous deployment with custom containers in Azure App Service," July 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/app-service/deploy-ci-cd-custom-container?tabs=acr&pivots=container-linux>. [Accessed October 2023].
- [22] Microsoft Azure, "Azure App Service plan overview," May 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/app-service/overview-hosting-plans>. [Accessed October 2023].
- [23] Microsoft Azure, "App Service pricing," [Online]. Available: <https://azure.microsoft.com/en-us/pricing/details/app-service/windows/>. [Accessed October 2023].
- [24] Microsoft Azure, "Azure Functions," [Online]. Available: <https://azure.microsoft.com/en-us/products/functions>. [Accessed October 2023].
- [25] Microsoft Azure, "Azure Functions triggers and bindings concepts," June 2023. [Online]. Available: <https://learn.microsoft.com/en-GB/azure/azure-functions/functions-triggers-bindings?tabs=isolated-process%2Cpython-v2&pivots=programming-language-python>. [Accessed October 2023].
- [26] Microsoft Azure, "What is Azure Static Web Apps?," April 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/static-web-apps/overview>. [Accessed October 2023].
- [27] React, [Online]. Available: <https://legacy.reactjs.org/>. [Accessed October 2023].
- [28] Hashicorp, "Modules," [Online]. Available: <https://developer.hashicorp.com/terraform/language/modules>. [Accessed October 2023].
- [29] Google Cloud, "Best practices for using Terraform," [Online]. Available: <https://cloud.google.com/docs/terraform/best-practices-for-terraform>. [Accessed October 2023].

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

Hamburg, _____