DEPLOYMENT ANALYSIS OF SERVERLESS AND NON-SERVERLESS HOSTING INFRASTRUCTURE WITH SAAS IMPLEMENTATION OF AN E-COMMERCE WEBSITE.

BY

OGHOSA BENJAMIN EDOMA

ENG1703941

500 LEVEL

DEPARTMENT OF COMPUTER ENGINEERING,
UNIVERSITY OF BENIN

SUPERVISED BY ENGR DR ISI EDEOGHON

SUBMITTED TO

DEPARTMENT OF COMPUTER ENGINEERING,
UNIVERSITY OF BENIN

SEPTEMBER 2023

CERTIFICATION

This is to clarify that the research work titled **"Deployment Analysis Of Serverless And Non-Serverless Hosting Infrastructure With SaaS Implementation Of An E-Commerce Website"** carried out by **Edoma Oghosa Benjamin** of Matric Number **ENG1703941** of the Department of Computer Engineering, has been completed, successfully passed the anti-plagiarism test requirement and submitted to the Department of Computer Engineering, University of Benin in partial fulfilment of the requirements for the award of Bachelor of Engineering (B.ENG) in Computer Engineering.

....

ENGR. DR. ISI EDEOGHON Date

(Project Supervisor)

....

ENGR. DR. (Mrs.) O. OKOSUN

(Head of Department) Date

DEDICATION

I humbly dedicate this work to God Almighty, acknowledging His immeasurable blessings and guidance that have sustained me throughout the process of this research and to my mom for her love and support all these years

ACKNOWLEDGEMENT

First and Foremost I would like to thank Mom, Miss Ihenyen M.I for the unwavering support, prayer and guidance which held me through all these years and helped me get to this point where I am. Without your efforts, none of this would have been possible, I am truly grateful for your contributions. And to my siblings, I would like to say thank you for your love and support throughout this period

I would like to extend my sincere appreciation to my project supervisor, Engr Dr Isi Edeoghon for his guidance, push, and unwavering support. Your expertise and insights have shaped my project and pushed me to excel. I am truly grateful for the opportunity to learn from you and for your belief in my abilities.

I would also like to acknowledge the invaluable contributions of the entire staff members of Computer Engineering whose collective efforts and expertise have impacted me, I say big a thank you to everyone.

To my entire family who have always been with me along the way, I would like to say thank you for the love and support.

And to my friends, Okhamena Azeez Kehinde and Aziken Jefferson, I would like to say a big thank you for your words of wisdom and advice during the period of this project.

To every one of you, thank you for your prayers, financial support, guidance, encouragement, and commitment. I am forever grateful for your presence in my life and for the part you have played in the success of my project.

List of Figures

List of Plates

ABSTRACT

This project presents a comprehensive analysis of the deployment strategies involving serverless and non-serverless hosting infrastructure within the context of Software as a Service (SaaS) implementation. The rapid advancements in cloud computing have introduced new paradigms for hosting applications, and the comparison of serverless and non-serverless(traditional) hosting approaches has gained significant attention in recent years. This study aims to evaluate the performance, scalability, cost-efficiency, and resource utilization of both serverless and non-serverless architectures in the context of deploying SaaS applications with an implementation in the form of an e-commerce site.

The research methodology encompasses a series of experiments conducted on real-world scenarios using popular cloud platforms. Performance metrics, such as response time, throughput, and scalability, are carefully measured and analyzed. Additionally, the consumption of computing resources and associated costs are thoroughly assessed to provide a comprehensive view of the two hosting infrastructures. The trade-offs between the two approaches are discussed, and guidelines are provided to aid decision-making processes when selecting the most appropriate hosting infrastructure for specific SaaS applications.

The findings indicate that serverless hosting exhibits several advantages in terms of auto-scalability, reduced operational complexity, and cost-effectiveness for applications with varying workloads. On the other hand, non-serverless hosting demonstrates better performance in scenarios with predictable and consistent demands.

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND OF STUDY

Non-serverless hosting infrastructure refers to traditional hosting methods where applications are deployed and run on dedicated servers or virtual machines without leveraging serverless computing platforms. This type of infrastructure requires manual provisioning, configuration, and management of servers and networking components.

The history of non-serverless hosting infrastructure dates back to the early days of the Internet. In the late 1990s, web hosting companies began offering shared hosting services, which allowed multiple websites to be hosted on a single server. This was followed by the introduction of virtual private servers (VPS) in the early 2000s, which allowed customers to have their dedicated server resources. In the mid-2000s, cloud computing emerged as a viable alternative to traditional hosting solutions, allowing customers to access computing resources on demand and pay only for what they use. This ushered in an era of Infrastructure as a Service (IaaS) providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). These providers offer a wide range of services including computing, storage, networking, and databases that can be used to build and deploy applications without having to manage physical servers.

Serverless hosting infrastructure refers to a cloud computing model where applications are deployed and executed without the need for managing or provisioning servers. The infrastructure is provided and managed by cloud service providers, and developers can focus solely on writing code for their applications without worrying about server management.

The concept of serverless hosting infrastructure has been around since the early 2000s, but it wasn't until the mid-2010s that it began to gain traction. In 2014, Amazon Web Services (AWS)

launched its Lambda service, which allowed developers to run code without having to manage servers. This was followed by Google Cloud Functions in 2016 and Microsoft Azure Functions in 2017. Since then, serverless hosting infrastructure has become increasingly popular as a way to reduce costs and simplify development processes. It is now used by many companies for a variety of applications, from web applications to data processing pipelines. Serverless hosting infrastructure is also being used for machine learning and artificial intelligence applications, as well as for IoT solutions(**Ivan. C, et, al**)

1.2 Statement of the problem

In today's rapidly evolving digital landscape, e-commerce is a dominant force in the global economy. With the growth of online shopping, businesses are continually seeking innovative ways to deliver seamless and cost-effective e-commerce solutions to their customers. The choice of hosting infrastructure for an e-commerce website is a critical decision, as it directly impacts factors such as scalability, performance, cost efficiency, and overall customer experience. This project aims to address the complex challenges associated with hosting infrastructure selection by conducting a comprehensive deployment analysis of serverless and non-serverless hosting options with a Software as a Service (SaaS) implementation for an e-commerce website.

1.3 Aim and Objectives

Aim: Deployment analysis of serverless and non-serverless hosting infrastructure with SaaS implementation

Objectives:

The objectives of the project on the deployment of serverless and non-serverless hosting infrastructure are to provide a step-by-step guide for implementing the project. The objectives of the project include:

i. Obtain the deployment steps for serverless infrastructure
ii. Obtain the deployment steps for non-serverless infrastructure

iii. Develop an E-commerce Website
iv. Deploy the E-commerce Website using Serverless Deployment
v. Analyze the performance, cost implications and various factors affecting both the serverless and non-serverless hosting infrastructure.

1.4 Scope of project

The project work will not go into the analysis of the complexities of the deployment of serverless and non-serverless hosting infrastructure, however, this project will be focused on the details of both topics and the serverless hosting will be implemented by building an e-commerce website and deploying it to the cloud.

1.5 Relevance of The Project

TECHNICAL KNOW-HOW ON DEPLOYMENT:

It is one thing to know what to do, it is another thing to know how to do it. This project will teach such individuals how to make their applications available to anybody anywhere in the world and the steps to take when deploying their application irrespective of the method chosen.

BETTER DECISION-MAKING ON THE METHOD OF DEPLOYMENT

This project will help developers or individuals who are planning on deploying their application make better decisions on the type of deployment to opt for depending on the individual needs or wants

CHOOSING THE MOST ECONOMICAL METHOD

This project will help individuals choose the most economical option for deployment whether it is serverless or non-serverless, i.e. the option that the individual budget can afford and the option that would save the individual less stress.

CHAPTER TWO

LITERATURE REVIEW

2.1 Theoretical Framework of Deployment Analysis Of Serverless And Non-Serverless Hosting Infrastructure With SAAS Implementation Of An E-Commerce Website

The typical framework for this project would involve obtaining the steps for both serverless and non-serverless deployment along with the SAAS implementation of a serverless E-Commerce website to further demonstrate the implementation of the serverless architecture.

2.2 Review of Key Concepts

The key concepts used in this project such as serverless deployment, non-serverless deployment, SaaS are described in this section.

2.2.1 Serverless Deployment

It is a development model built for creating and running applications without the need for server management. They are provisioned, maintained, and scaled by a third-party cloud provider, while the developers just write and deploy the code. It is a cloud computing model that allows developers to run code without managing the underlying server infrastructure. In a serverless deployment, developers write and upload functions or code snippets to a cloud provider's platform, and the cloud provider automatically manages the infrastructure needed to execute that code.

Services rendered by serverless deployment include:

- **Automatic Scaling:** Serverless platforms automatically scale the application based on demand. It dynamically allocates resources to handle incoming requests, ensuring that the application can handle varying workloads without manual intervention.

- **Pay-as-you-go Pricing:** Serverless deployment follows a pay-as-you-go model, where you are billed based on actual usage. You are charged for the number of function executions, compute time, and resource consumption, making it cost-effective for applications with varying traffic patterns.

- **Event-Driven Architecture:** Serverless functions are triggered by events, such as HTTP requests, database changes, file uploads, or time-based schedules. This event-driven architecture allows for real-time processing and event-based workflows.

- **Focus on Code:** Serverless platforms abstract away much of the infrastructure management, allowing developers to focus primarily on writing application code. This streamlines the development process and reduces operational overhead.

- **No Server Management:** With serverless deployment, developers do not need to provision or manage servers. The platform handles all the server management tasks, including scaling, patching, and infrastructure maintenance.

- **High Availability:** Serverless platforms typically provide built-in high availability. Functions are automatically replicated across multiple data centres, ensuring that the application remains accessible even in the case of hardware failures.

- **Seamless Integration with Other Services:** Serverless platforms integrate well with other cloud services, such as databases, storage solutions, messaging systems, authentication services, and more. This allows developers to build feature-rich applications by leveraging existing services.

- **Rapid Deployment:** Serverless applications can be deployed quickly and easily. Developers can update and redeploy functions without worrying about managing server instances.

- **Scalability:** Serverless architectures inherently support auto-scaling, allowing applications to handle sudden spikes in traffic without manual intervention. This makes it suitable for applications with unpredictable or highly variable workloads.

- **Lower Infrastructure Cost:** Since serverless platforms automatically manage the infrastructure and resources, it reduces the need for overprovisioning and minimizes idle resource costs. This can lead to cost savings for many use cases.

- **Developer-Friendly Ecosystem:** Serverless platforms often provide extensive tooling, documentation, and development environments that facilitate rapid prototyping and development iterations.

Serverless deployment is based on two main components: Function as a Service (FaaS) and Backend as a Service (BaaS).

FaaS is a computing service that is necessary to run stand-alone pieces of code known as functions in the cloud. They are small, short-lived, serve only one purpose, and remain inactive until some event triggers them. Function code is stored in the cloud, but the instance that runs it disappears once the function finishes its task.

BaaS is a cloud computing service that allows software developers to focus on the front end while the back end, implemented by someone else, takes place on remote servers. BaaS can include such things as authentication, storage and geolocation services, user management, etc.

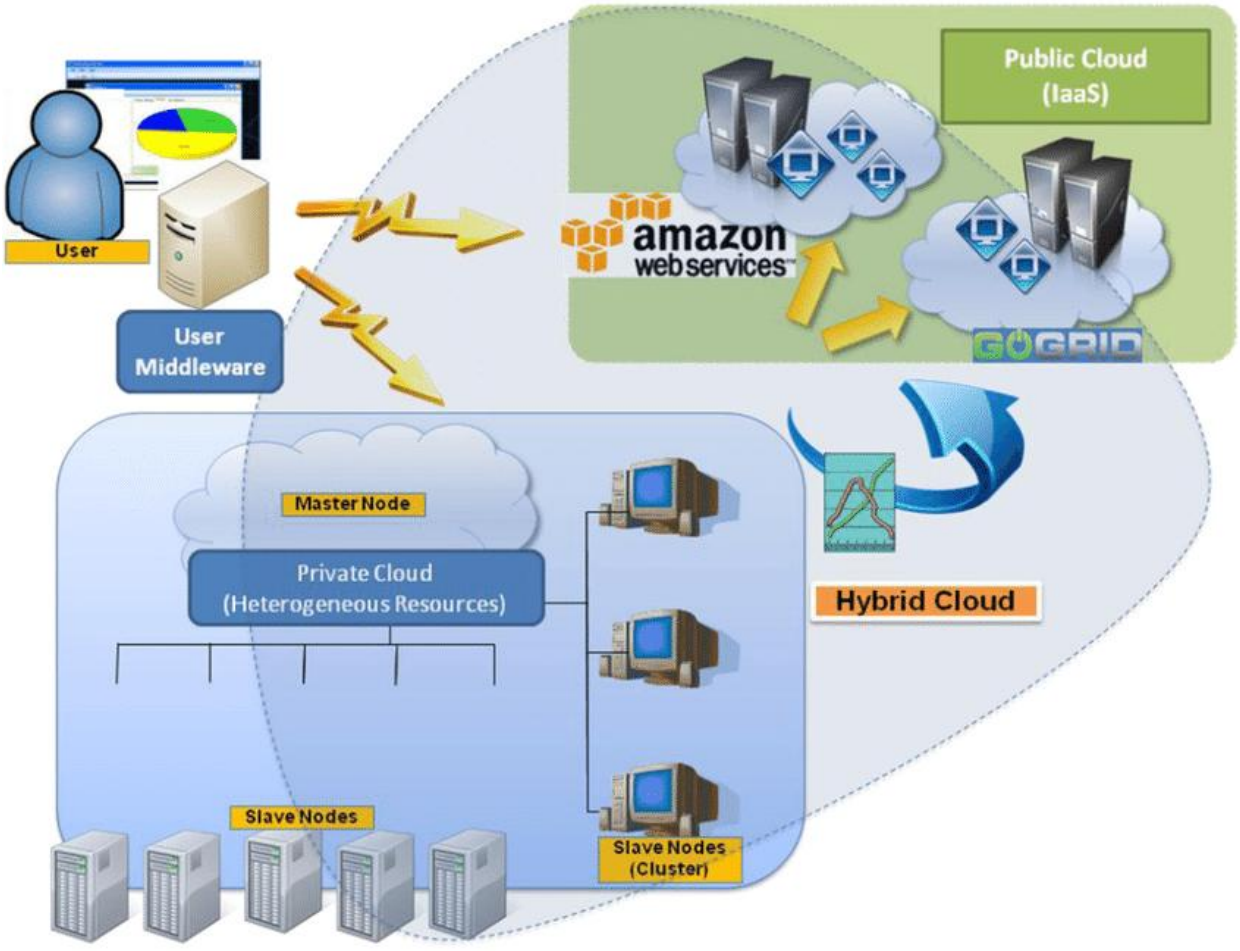

Fig 2.1 A typical Serverless deployment Architecture

2.2.2 GitHub

GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code. To understand exactly what GitHub is, you need to know two connected principles:

- Version control
- Git

Version control helps developers track and manage changes to a software source code. As a software project grows, version control becomes essential. For example, If a core developer wanted to work on one specific part of a large e-commerce codebase, it wouldn't be safe or efficient to have them directly edit the "official" source code.

Instead, version control lets developers safely work through branching and merging. With branching, a developer duplicates part of the source code (called the repository). The developer can then safely make changes to that part of the code without affecting the rest of the project.

Then, once the developer gets his or her part of the code working properly, he or she can merge that code back into the main source code to make it official. All of these changes are then tracked and can be reverted if need be.

Git is a specific open-source version control system which was created by Linus Torvalds in 2005. Specifically, Git is a distributed version control system, which means that the entire codebase and history are available on every developer's computer, which allows for easy branching and merging.

GitHub is a for-profit company that offers a cloud-based Git repository hosting service. Essentially, it makes it a lot easier for individuals and teams to use Git for version control and collaboration. Developers can easily deploy their applications to various cloud platforms by linking the respective repository hosted in GitHub to the cloud platform.

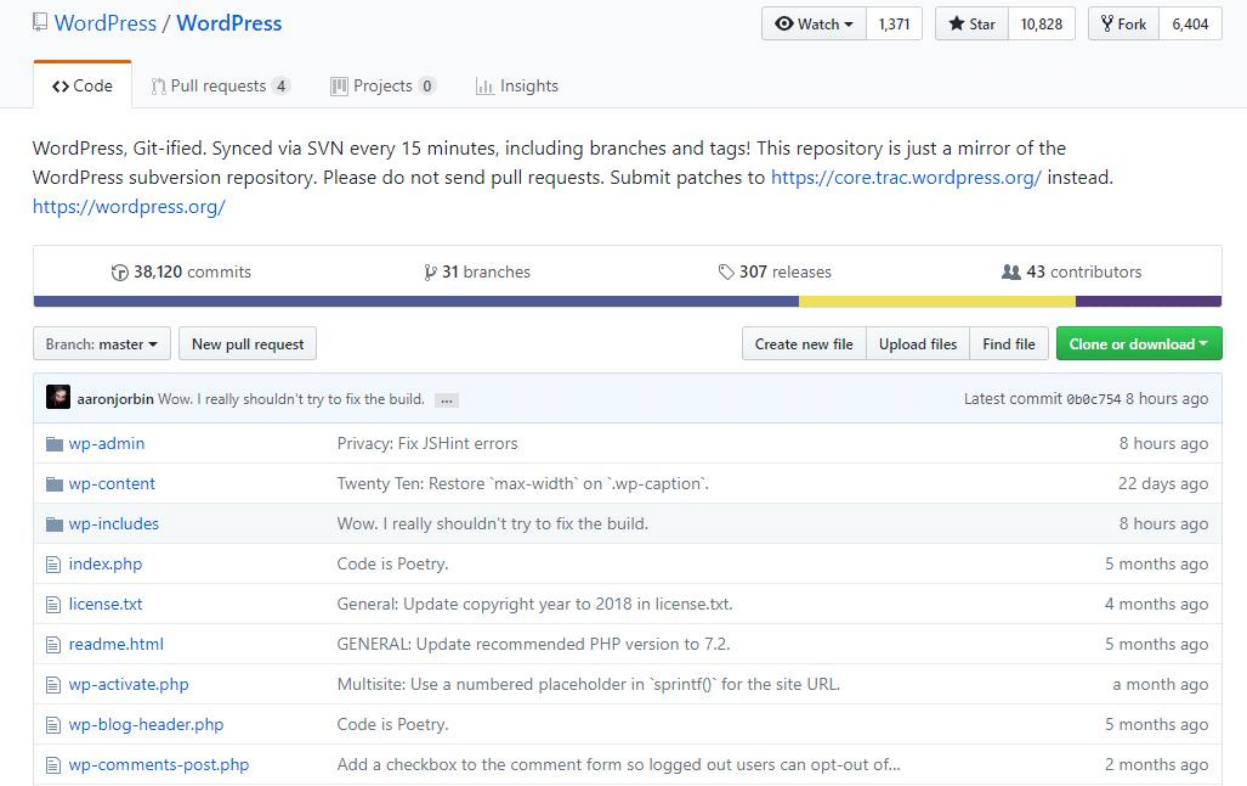

Fig 2.2 Snapshot showing a repository hosted on GitHub

2.2.3 Non-Serverless Deployment

Non-serverless deployment, also known as traditional or traditional server-based deployment, refers to the conventional cloud computing model where developers are responsible for provisioning, managing, and maintaining the underlying server infrastructure on which their applications run. In contrast to serverless deployment, where developers only need to focus on writing code and defining functions, non-serverless deployment involves more hands-on management of servers, virtual machines (VMs), containers, and other infrastructure components.

2.2.4 VMs (Virtual Machines):

Virtual machines (VMs) are software-based simulations of computer systems. They enable the running of numerous virtual instances of operating systems and programs on a single physical

server. Each VM is managed and allocated resources by hypervisor software, which isolates them from each other and the underlying hardware. VMware, Hyper-V, and KVM (Kernel-based Virtual Machine) are popular hypervisors.

VMs improve resource usage by allowing numerous VMs to share the same physical server, lowering hardware costs and energy consumption. Virtual machines provide tight separation between distinct applications or services, preventing one VM from interfering with others. VMs are extremely portable and can be simply moved between physical servers or cloud settings. VMs are appropriate for applications with varied resource requirements or operating systems, allowing developers to create hybrid applications

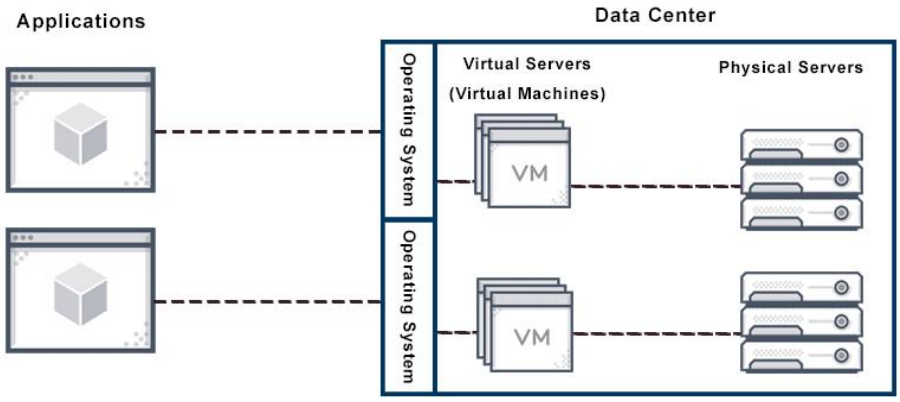

Figure 2.3 Architecture of a Virtual Machine

2.2.5 Physical Servers.

These are Servers that are physically present. The classic hardware-based solution is physical servers, in which a single server is dedicated to running an operating system and hosting one or more applications. Each physical server is self-contained, with no resources shared with other servers. For resource-intensive applications that require specialized hardware, physical servers can provide greater performance and are easier to manage and configure for unique purposes. Physical servers are useful in instances such as for outdated systems or where rigorous hardware

control is required. Physical servers are often employed for applications that require predictable and reliable workloads, as well as isolation and dedicated resources.

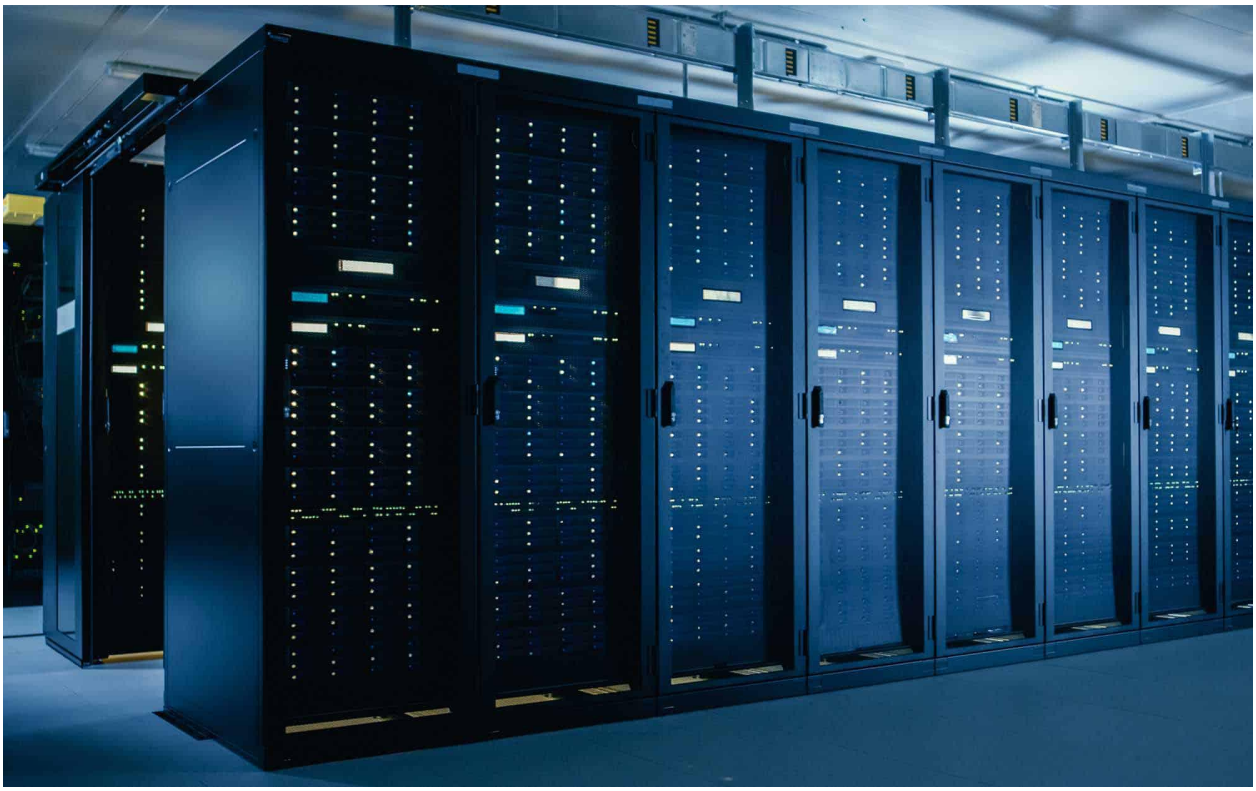

Figure 2.4 Picture showing multiple physical servers

2.2.6 E-Commerce Web-application

An e-commerce web application is a type of online platform that allows businesses to sell products or services to customers over the Internet. It provides a virtual storefront where users can browse, search, and purchase items from the comfort of their own devices. key features and components of an e-commerce web application include:

- Product Catalog: The application showcases a catalogue of products or services with detailed information such as product images, descriptions, prices, and availability.

- Shopping Cart: Users can add items to their virtual shopping cart while browsing the product catalogue. The cart keeps track of selected items until the user proceeds to checkout.
- Inventory Management: The application keeps track of product inventory levels to prevent overselling and provide accurate stock information to customers.
- Responsive Design: E-commerce web applications are designed to be responsive and mobile-friendly, ensuring a seamless shopping experience across various devices, including smartphones and tablets.
- Promotions and Discounts: The application may offer promotional campaigns, discount codes, and special offers to encourage.

An E-commerce web application can be developed using various programming languages such as HTML, CSS, JavaScript, and TypeScript or frameworks such as React.JS, and Next.JS.

Figure 2.5 A snapshot showing an e-commerce website accessed from devices through the internet

2.2.7 Web-application Development Technologies

There are various technologies used in developing an E-Commerce web application. Some of these technologies include.

1. Programming languages

A programming language is a formal system designed to instruct a computer to perform specific tasks. It serves as a medium of communication between humans and computers, enabling programmers to write code that the computer can understand and execute. Programming languages are used to develop software applications, websites, and various other computer programs. They come in different types, each with its syntax and set of rules for writing code. Some well-known programming languages include:

i. Python: Python is a high-level, versatile, and easy-to-read language known for its simplicity and readability. It is widely used in web development, data science, artificial intelligence, automation, and more.

ii. JavaScript: JavaScript is a scripting language used primarily for front-end web development, adding interactivity and dynamic behaviour to web pages.

iii. TypeScript: TypeScript is a superset of JavaScript that adds static typing to the language, making it easier to manage large-scale JavaScript projects.

2. Web-development Frameworks

Web development frameworks are software tools that provide a foundation and structure for building web applications. They offer pre-defined components, libraries, and patterns that streamline the development process, making it faster and more efficient. Web development

frameworks can be categorized into frontend (client-side) and backend (server-side) frameworks. Here are some popular examples of each:

Frontend (Client-Side) Web Development Frameworks:

i. React: A JavaScript library for building user interfaces, maintained by Facebook. It is component-based, making it easy to create reusable UI components.

ii. Next.JS: Next.js is a popular open-source framework for building modern web applications and websites with React. It is designed to simplify the development of server-side rendered (SSR) React applications, making it easier for developers to create fast, dynamic, and SEO-friendly web pages. Next.js is built on top of React and adds powerful features and optimizations to enhance the developer experience and improve performance.

Backend (Server-Side) Web Development Frameworks:

i. Express.js: A minimalist and flexible Node.js framework for building web applications and APIs.

ii. Django: A high-level Python web framework that encourages rapid development and follows the "batteries-included" philosophy.

These frameworks save developers time by abstracting away common tasks and providing a standardized structure, allowing them to focus more on application logic and less on boilerplate code. The choice of framework often depends on the programming language, specific project requirements, developer expertise, and personal preferences. Additionally, some full-stack frameworks, like MEAN (MongoDB, Express.js, Angular, Node.js) and MERN (MongoDB, Express.js, React, Node.js), combine frontend and backend technologies into a cohesive package for building end-to-end web applications.

3. Database component

In web development, a database component plays a crucial role in storing, managing, and retrieving data for web applications. It serves as a central repository for organizing and persistently storing various types of information related to the application, its users, and its content. The database component enables web developers to create dynamic, data-driven applications that can handle user interactions, process transactions, and deliver personalized content. Here are some key uses of the database component in web development. An Example of a database component is MongoDB.

MongoDB: is a popular open-source NoSQL database management system designed to handle unstructured or semi-structured data. It belongs to the family of document-oriented databases and is known for its flexibility, scalability, and ease of use. MongoDB stores data in JSON-like BSON (Binary JSON) format, making it ideal for applications with evolving and dynamic data structures.

2.2.8 Operating System

An operating system (OS) is a fundamental software component that manages and controls a computer's hardware and software resources, facilitating user interaction and the execution of applications. It serves as an intermediary layer between the user and the computer's hardware, providing essential services such as process management, memory management, file system management, and hardware-device interaction. The primary functions of an OS include managing system resources, scheduling tasks, providing a user interface, ensuring security, and enabling efficient communication between software applications and hardware components. Operating systems are essential for the proper functioning of computers and a wide range of computing devices, from personal computers and servers to mobile devices and embedded systems.

Since the developer is responsible for server management in non-serverless architecture, the developer must select a suitable operating system that will meet and satisfy the application requirements.

2.2.9 Software as a Service (SAAS)

Software as a service (SaaS) allows users to connect to and use cloud-based apps over the Internet. Common examples are email, calendaring, and office tools (such as Microsoft Office 365).

SaaS provides a complete software solution that you purchase on a pay-as-you-go basis from a cloud service provider. You rent the use of an app for your organization, and your users connect to it over the Internet, usually with a web browser. All of the underlying infrastructure, middleware, app software, and app data are located in the service provider's data centre. The service provider manages the hardware and software, and with the appropriate service agreement, will ensure the availability and security of the app and your data as well. SaaS allows an organization to get quickly up and running with an app at minimal upfront cost.

Advantages of SaaS include gaining access to sophisticated applications, paying only for what you use, using free client software, easily mobilization of workforce, and accessing app data from anywhere.

2.2.10 FTP (File Transfer Protocol)

FTP (File Transfer Protocol) is a network protocol for transmitting files between computers over Transmission Control Protocol/Internet Protocol (TCP/IP) connections.

FTP is a client-server protocol that relies on two communication channels between the client and server: a command channel for controlling the conversation and a data channel for transmitting file content.

FTP sessions work in active or passive modes:

Active mode. After a client initiates a session via a command channel request, the server creates a data connection back to the client and begins transferring data.

Passive mode. The server uses the command channel to send the client the information it needs to open a data channel. Because passive mode has the client initiating all connections, it works well across firewalls and network address translation gateways.

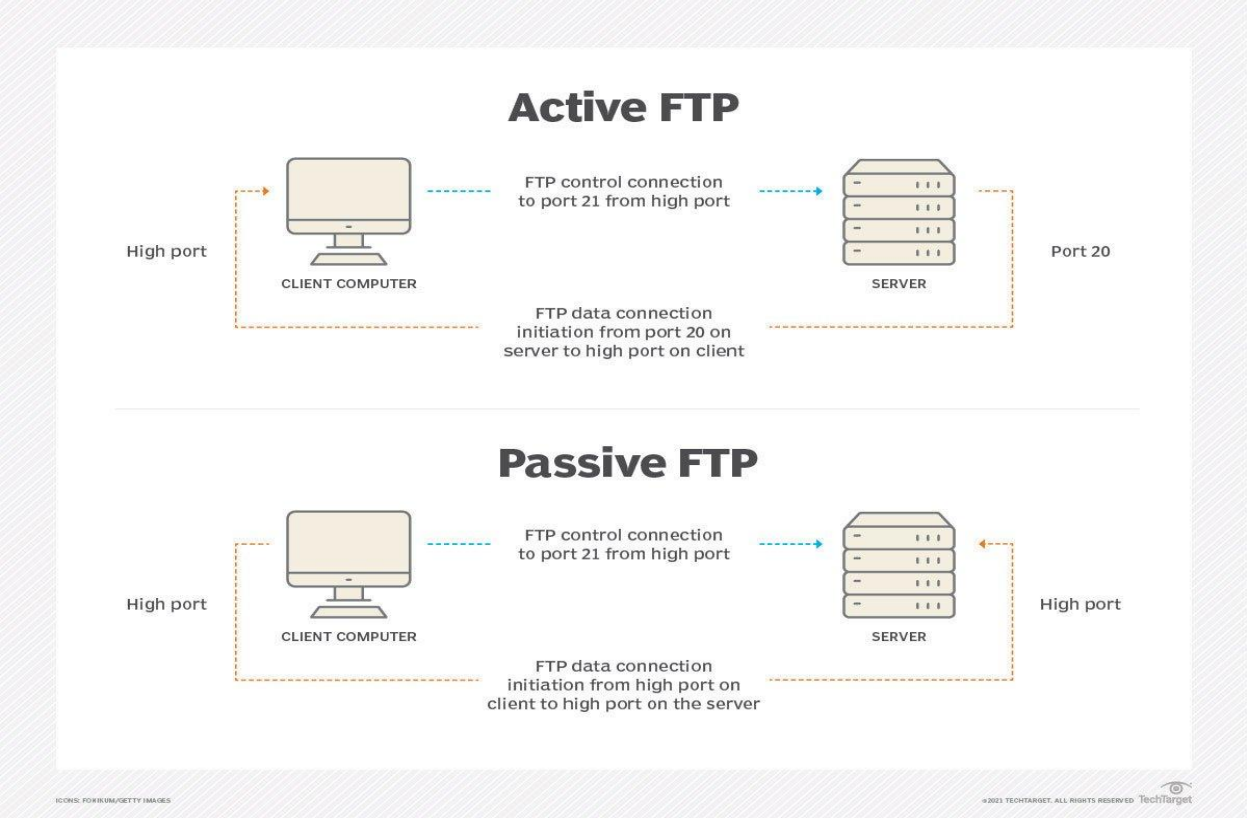

Figure 2.6 Architecture of Active and Passive FTP

2.2.11 Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is the foundation of the World Wide Web and is used to load webpages using hypertext links. HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. A typical flow over HTTP involves a client machine requesting a server, which then sends a response message.

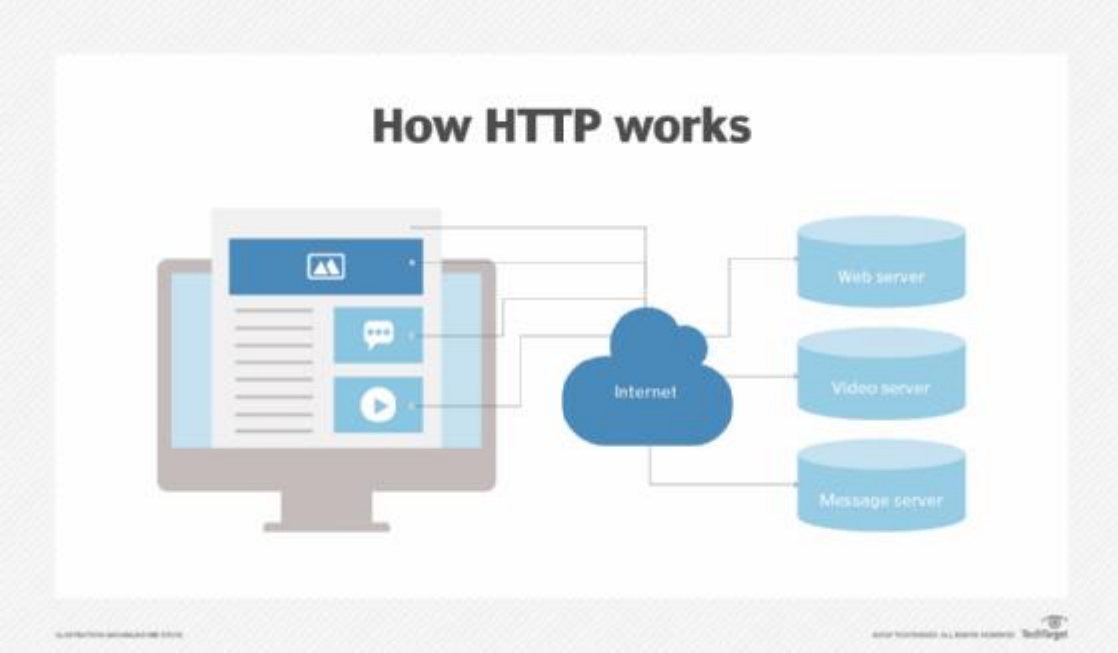

Figure 2.7 Snapshot showing how HTTP works

2.2.12 Virtual Network Computing (VNC)

Virtual Network Computing (VNC) is a free tool that allows a client to connect to a server, and interact with the desktop of the remote machine. Virtual Network Computing, or VNC, is an open-source application that provides screen-sharing services and is available for virtually all operating systems such as Windows, Linux, and of course OS X.

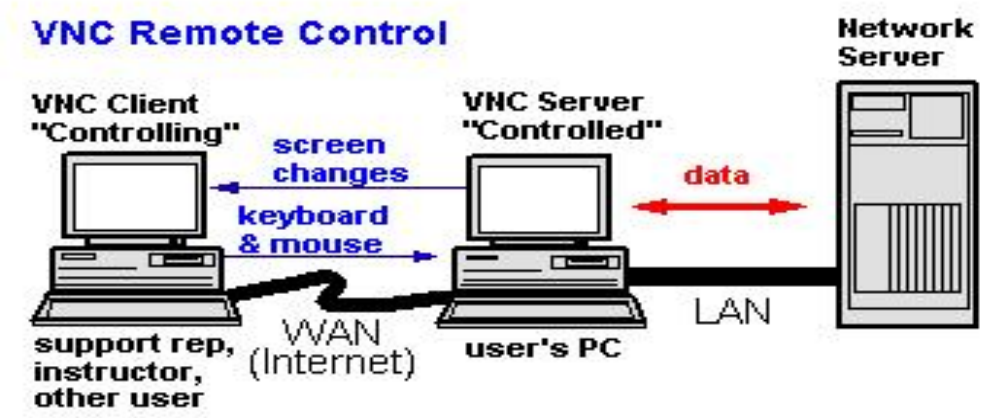

Figure 2.8 Snapshot showing how Virtual Network Computing (VNC) works

2.3 Review of Related Works

Admittedly, much work has been done in the area of serverless and non-serverless deployment and these works are too numerous for a comprehensive listing. Consequently, key examples will be provided that will be used as a reference for the development of this project. For example,

Hassan B. Hassan, Saman A. Barakat & Qusay I. Sarhan (2021) work on "Survey on serverless computing", they performed a systematic survey on 275 research papers that examined serverless computing, the research papers were from well-known literature databases and they were extensively reviewed to extract useful data. Then, the obtained data were analyzed to answer several research questions regarding the state-of-the-art contributions of serverless computing, its concepts, its platforms, and so on.

In **Gojko Adzic and Robert Chatley** work on "**Serverless Computing: Economic and Architectural Impact(2017)"**, they analysed the impact of migrating an application to a cloud platform, the result showed how migrating an application to the Lambda deployment architecture reduced hosting costs - by between 66% and 95%. One drawback of this application was that they only took hosting costs into account and not other factors such as performance, latency and speed.

Shakirat Haroon- Sulyman (2014) work on the **"Client-Server Model"**. This study identified the various Client/Server technologies that are available, explained how the technologies are used, and compared and contrasted these various technologies in the Client/Server model. The study utilized a simulated e-commerce online book-order model, The drawback of this study was that it only utilized a simulated e-commerce online book-order and not an actual e-commerce online book-order model.

Dothang Truong work on **"How cloud computing enhances competitive advantages: A research model for small businesses (2010)"** This study developed a research model of cloud

computing from a managerial perspective and focused on small businesses. A resource-based view theory was used to propose a research model which explores the influence of cloud computing-related resources on a small business' competitive advantages. One limitation of this study was not exploring the influences of non-serverless infrastructure on small businesses

Ioana Baldini et al (2017) work on **"Serverless Computing: Current Trends and Open Problems"**, they analysed existing serverless platforms from industry, academia, and open-source projects. The study identified key characteristics and use cases, and described technical challenges and open problems. This study did not implement a practical architecture with the serverless infrastructure

CHAPTER THREE

RESEARCH METHODOLOGY

This chapter outlines the procedures and methods necessary in implanting both the serverless and non-serverless deployment architecture. The approach for the serverless architecture is outlined and explained in great detail in Section 3. 1 The approach for implementing the non-serverless architecture is likewise outlined and explained in Section 3.2.

Both the Serverless and Non-Serverless infrastructure are implemented in the context of deploying an e-commerce website, as this will further explain both approaches better and gives us a better understanding of how both infrastructure works.

3. 1 Workflow Process For Serverless Deployment

This section overviews the Serverless infrastructure implementation. The workflow needed to achieve the serverless deployment is outlined and explained in detail.

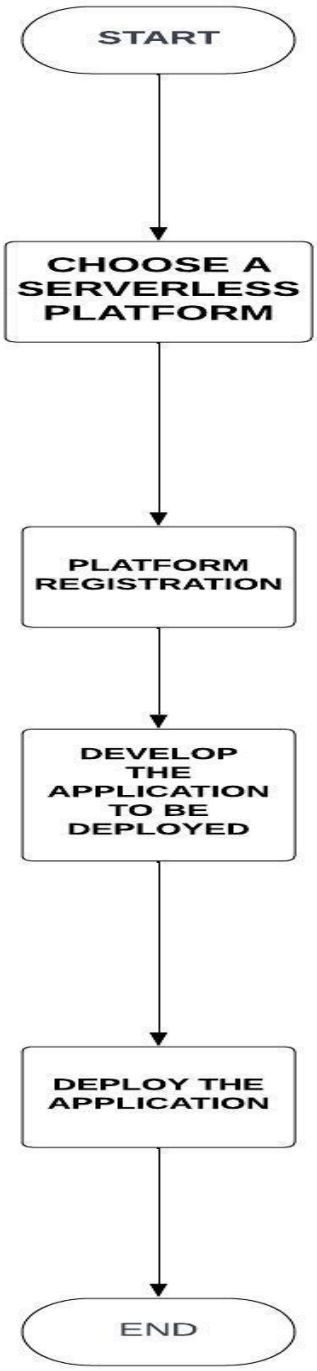

Figure 3.1 Workflow diagram for Serverless hosting

3. 1. 1 CHOOSING A SERVERLESS PLATFORM

Choosing a serverless platform is a crucial step, as it sets the foundation for the entire serverless deployment. A serverless platform provides the necessary infrastructure and tools to deploy and manage applications without worrying about traditional server management.

The serverless platform used for the deployment was Vercel and the following factors were considered before choosing the platform.

- **Language Support:** Checking which programming languages are supported by the serverless platform is vital. Most platforms offer support for popular languages like JavaScript, Python, Java, Go, and more. This is to ensure that the platform supports the language that suits the project's specific needs. The main Programming language used to develop the application to be deployed was JavaScript and a framework called Next.JS, the selected cloud platform supported these technologies.

- **Function Execution Time and Limits:** Different serverless platforms impose limits on the execution time and resources allocated to a function. For example, vercel has a maximum execution time of 30 seconds (to begin returning a response, but can continue streaming data.), while other providers might have different limits. The application's requirements were analyzed to ensure they fit within the platform's constraints is very crucial.

- **Scalability and Concurrency:** Serverless platforms are known for their auto-scaling capabilities, which allow functions to automatically handle sudden spikes in traffic. The platform was examined on how it scaled with increased demand and how it handles concurrent executions. This was critical to ensure the application could handle varying workloads efficiently.

- **Event Triggers:** Serverless functions are typically event-driven, meaning they respond to specific events, such as HTTP requests, database changes, or file uploads. The available event triggers provided by the platform were evaluated to ensure they covered the use cases required by the application.

- **Integration with Other Services:** The efficiency of the serverless platform integration with other services offered by the cloud provider such as databases, storage, messaging services, and monitoring tool were analysed. This was to ensure that the platform had a Seamless integration with other services as this would simplify the development process and enhance the application's capabilities.

- **Vendor Lock-In and Portability:** The requirement to migrate the application to a different provider was considered so that the application would have better portability.

.

- **Pricing Model:** Since Cloud platforms charge based on function executions, compute time, and resource usage, the application's expected usage patterns were analysed to estimate cost-effectively.

3. 1. 2 PLATFORM REGISTRATION (CREATING AN ACCOUNT ON THE SERVERLESS DEPLOYMENT PLATFORM)

Once the perfect platform has been found, an account needs to be set up, this account could be a freemium or a premium account. A freemium account would be free to set up but might have certain restrictions. A premium account usually offer more services compared to a premium account and it usually has different pricing option, the higher the pricing option, the better the quality of services offered. A Premium account was selected because it offered better services.

3. 1. 3 DESIGN THE APPLICATION TO BE DEPLOYED

Designing the web application involves planning the user interface (UI) and user experience (UX) aspects before development and deployment, this sets the foundation for a user-friendly and visually appealing final product. The wireframes and mockups of the application were created to establish the application's visual elements, layout, and interactions. Figma (software used for interface design) was used to design the application interface and layout.

3. 1. 4 DEVELOP THE APPLICATION TO BE DEPLOYED

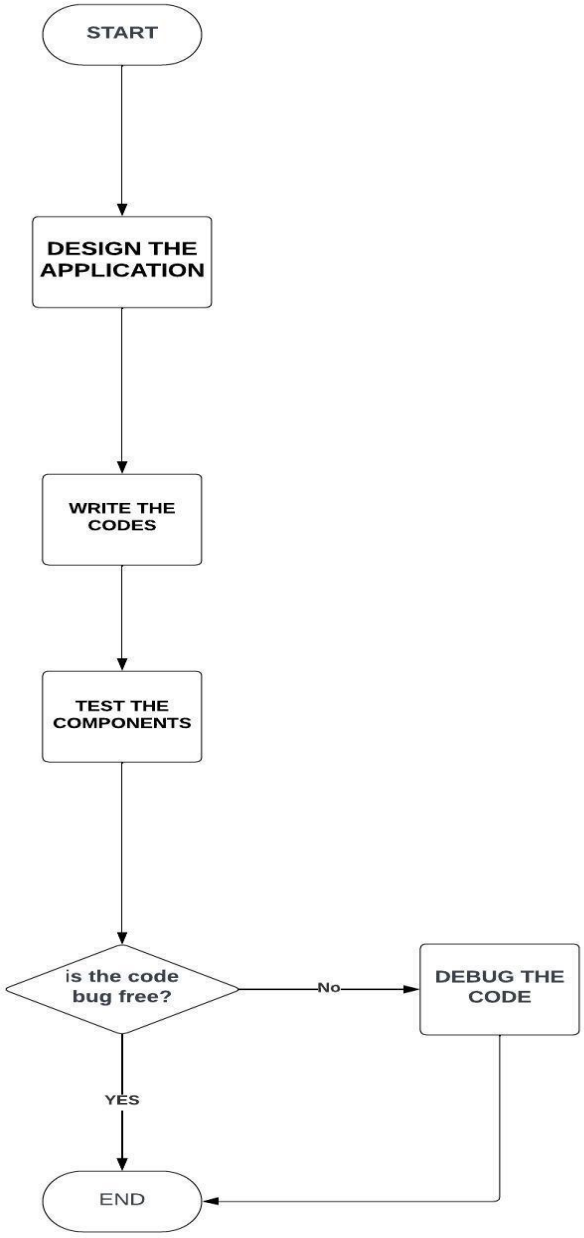

Figure 3.2. Workflow diagram for Developing the application/e-commerce website to be deployed

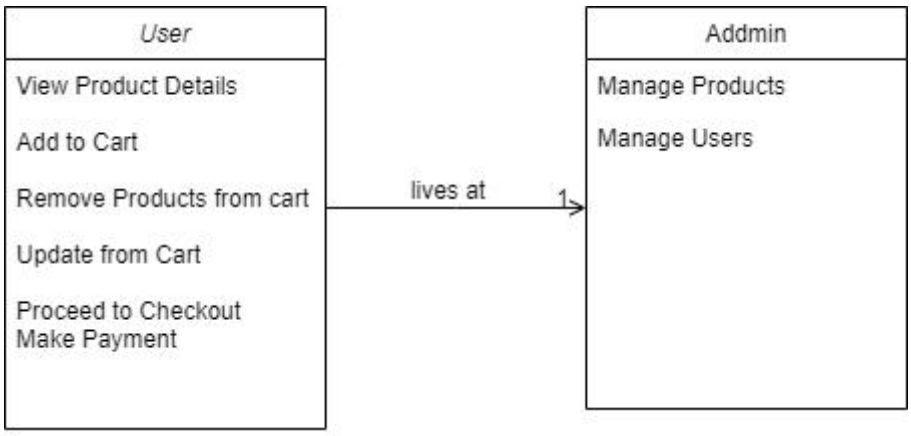

Figure 3.3 Use case diagram for the E-Commerce Website

Developing the application involved Front-end development Back-end development and database integration.

Front-end development involves writing the client-side code responsible for creating the user interface and handling user interactions. This includes HTML, CSS, and JavaScript programming to render the visual elements and enable dynamic behaviour. The client side of the application was developed primarily using a JavaScript framework called Next.JS, this involved developing the interface of the homepage, the products page, the cart interface and the checkout page as well as the necessary functions needed to interact with these pages. The homepage of the e-commerce website is the first page visited by a user when they click the link, a detailed description of each product is written on the product page and customers can the specific product to their cart from this page, the cart page shows the various products in the customer's cart that is the products the customers wishes to buy, these products are paid for by the customers in the checkout page the products are paid via online methods such as transfer, card payment e.t.c

Back-end development focuses on building the server-side components of the web application. This includes implementing server logic, handling data processing and storage, and setting up APIs and endpoints to communicate with the front end. Next.JS was used to develop the various API endpoints of the application.

The web application was connected to a chosen database or data storage system, whilst implementing the necessary data models and queries to interact with the database and manage data retrieval and manipulation. The database used in this scenario was MongoDB

3. 1. 5 TESTING THE APPLICATION

Various testing activities were carried out during development, such as unit testing to ensure individual components function correctly, integration testing to verify interactions between different parts of the application, and user acceptance testing (UAT) to involve end-users in validating the application against requirements.

3. 1. 6 DEPLOY THE APPLICATION

This required preparing the application for deployment to the chosen serverless platform. Which included packaging the code and any necessary dependencies, configuring environment variables, and setting up deployment scripts.

The web application was prepared for production use by setting up the production environment, configuring the application settings, and ensuring security measures were in place. Database migration was also necessary, and the code was compiled into a production-ready build. Testing in a staging environment helped in catching issues before deployment. The application was deployed to the cloud platform(in this case, Vercel) by uploading the source code to a GitHub account which is linked to a Vercel account, then the source code/application was finally deployed by selecting the repo(The GitHub folder contains the source code to be deployed) which was to be deployed from the Vercel account.

3. 2 Workflow Process For Non-Serverless Deployment

This section overviews the Serverless infrastructure implementation. The workflow necessary to achieve the serverless deployment is outlined and explained in detail.

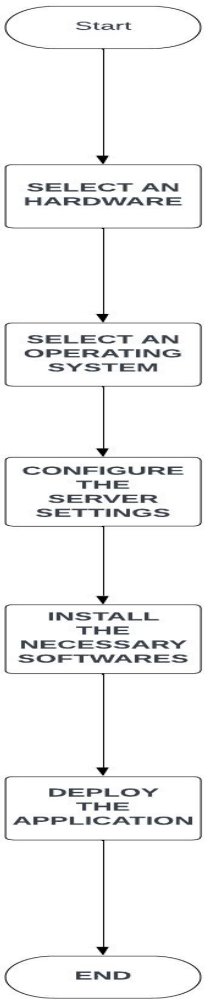

Figure 3.4 Workflow diagram for Non-Serverless hosting

3. 2. 1 Choosing Hardware:

The first step is choosing the hardware. This hardware could be a computer otherwise known as a dedicated computer. The computer needs to have enough storage space to house a 60 GB hard drive with at least 25 GB of free space, a dual-core processor of at least 2 GHz and 2 GB RAM,

a USB port and a DVD drive. The type of computer to use depends on certain factors such as the number of users, the need for storage space and the requirement of processing power.

3. 2. 2 Choose An Operating System (OS) And Install It

The second step is to choose an operating system. There are only two choices: Linux and Windows.

3. 2. 3 Set Up VNC

Virtual Network Computing (VNC) allows for remote access to a computer located elsewhere. Users can operate this computer just as if they were sitting right in front of it! All the resources that are available at the remote computer's location (printers, drives) are also available to the user who's accessing this computer, but anyone using the computer at the remote location can see what is happening. There are two main uses for VNC: to remotely access a work computer or to provide tech support remotely. A server can't function without this feature.

To access a computer through a VNC, the IP address (or the fully qualified domain name) of the computer that is been accessed needs to be known, as well as the password that's been designated in the VNC software.

The process of installing VNC it's quite simple. First, the vnc4server package is installed. Then configuration changes are made to the server.

3. 2. 4 Install File Transfer Protocol (FTP)

File Transfer Protocol (FTP) is a standard communication protocol that's used for file transfer from server to client on a computer network. After creating the server, A FTP server is a necessity. It is used to create a private cloud that's under the user's control and allows for the transfer of files at almost unlimited speeds. There are three types of FTP:

- FTP – Basic, unencrypted FTP, supported by most web browsers.

- FTPS – SSL/TLS encrypted FTP, which is widely used, even though it is not supported by major web browsers.
- FTPES – This version is upgraded to TLS/SSL encryption and is more firewall-friendly. Major browsers don't support it, but it's still a preferred way to establish a connection.

FTP can be installed by navigating to the Internet Information Services (IIS) Manager under the Control Panel and selecting the 'Add FTP Site' option. After configuring the FTP server the server can be activated

3. 2. 5 Install HTTP (Hypertext Transfer Protocol)

HTTP is installed to set up the hosting software. For the software to be fully operational, the specific LAMP software stack needs to be installed – Linux, Apache, MySQL, and PHP, which allows us to set up HTTP, use the server to communicate with databases, and store data. The HTTP (Hypertext Transfer Protocol) server is a powerful tool for production usage and it's simple enough to be used for local development, testing, and learning.

3. 2. 6 Deploy The Application

The application is deployed by transferring the application code and files to the server using FTP.

3.3 Analysis of the serverless and non-serverless infrastructure

This involves the analysis of both the serverless and non-serverless architecture. Both approaches were compared based on several factors and how the application behaved when deployed using either of these approaches. These factors include Resource management, deployment and development speed, Performance, Security, monitoring and debugging, Complexity, Vendor lock-in and Latency.

3.4 Software Development Life Cycle

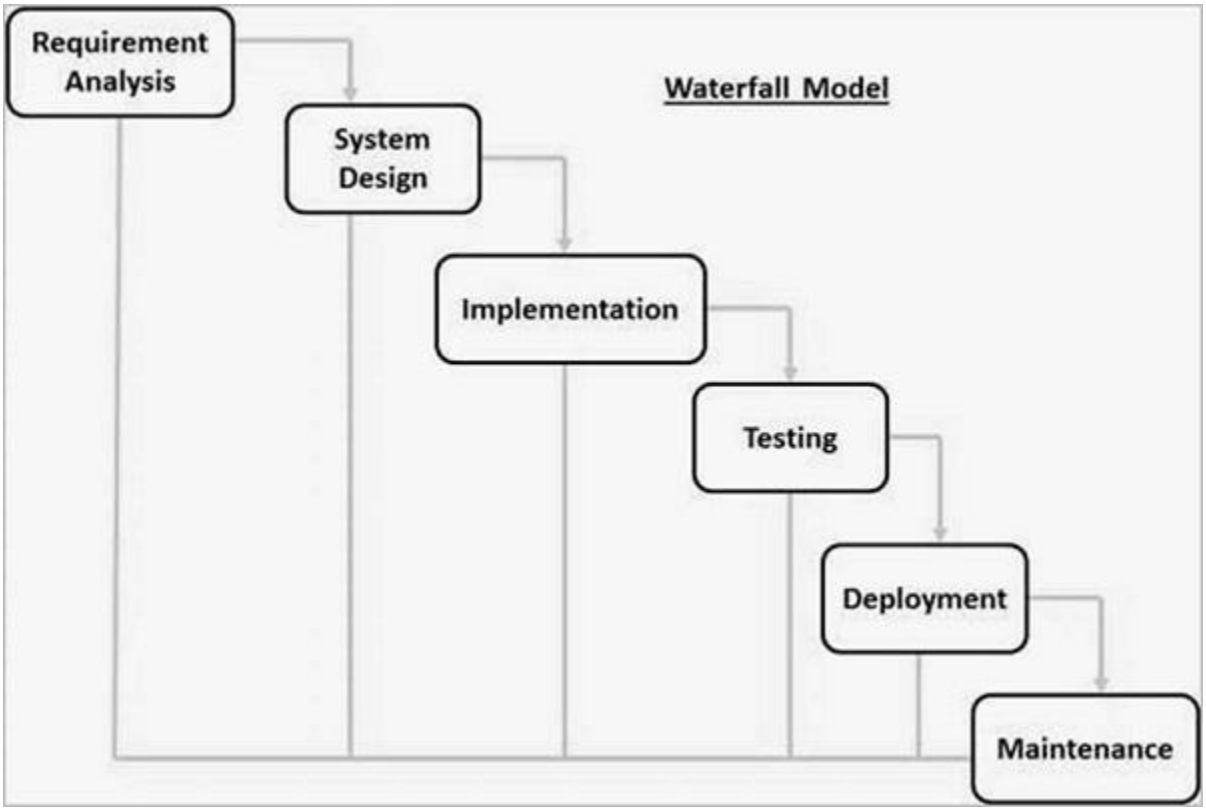

Figure 3.5 Software Development Life Cycle for the application

The software development life cycle model used for the application was the waterfall model.

Requirement Gathering and analysis − All possible requirements of the system to be developed were captured in this phase and documented in a requirement specification document.

System Design − The requirement specifications from the first phase were studied in this phase and the system design was prepared.

Implementation − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

Integration and Testing − All the units developed in the implementation phase are integrated into a system after testing each unit. Post integration the entire system is tested for any faults and failures.

Deployment of system − Once the functional and non-functional testing were done; the product was deployed in the customer environment or released into the market.

Maintenance − All issues which came up in the client environment were fixed and patches were released.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 RESULT PRESENTATION OVERVIEW

This chapter presents the results of the methods carried out in Chapter 3. An e-commerce website was developed and deployed to a cloud platform to demonstrate the serverless architecture, the implemented serverless architecture was compared with the non-serverless architecture based on various factors such as Resource management, deployment and development speed, Performance, Security, monitoring and debugging, Complexity, Vendor lock-in and Latency.

Section 4.2 presents an overview of the e-commerce website, with snapshots showing various pages of the e-commerce website.

Section 4.3 presents a results analysis on Resource management comparison between the Serverless and non-serverless architecture. Section 4.4 presents the analysis of the results of the Deployment and development speed comparison between the Serverless and non-serverless architecture. Section 4.5 presents the analysis of the results of the Performance comparison between the Serverless and non-serverless architecture. Section 4.6 presents the analysis of the results of the Security comparison between the Serverless and non-serverless architecture. Section 4.7 presents the analysis of the results of the complexity comparison between the Serverless and non-serverless architecture. Section 4.8 presents the analysis of the results of monitoring and debugging comparison between the Serverless and non-serverless architecture. Section 4.9 presents the analysis of the results of the Cost Model comparison between the Serverless and non-serverless architecture. Section 4.10 presents the analysis of the results of the Vendor lock-in comparison between the Serverless and non-serverless architecture. Section 4.11 presents the analysis of the results of the Latency comparison between the Serverless and non-serverless architecture

4.2 OVERVIEW OF THE E-COMMERCE WEBSITE

The e-commerce website developed is a web shop which sells phone cases. Since the e-commerce website is deployed using a cloud platform, it can be accessed by anyone with the website URL. The e-commerce website has various web pages such as the homepage which is the first page visited by the end-user when they visit the website, a product page which shows a detailed description of each product, a cart page where customers can see the various products that have been added to their cart(products the customers want to purchase), a checkout page where customers can pay for their orders, some functions used in the this website includes the add to cart function and a checkout function allowing customers to pay for their orders using online payment methods.

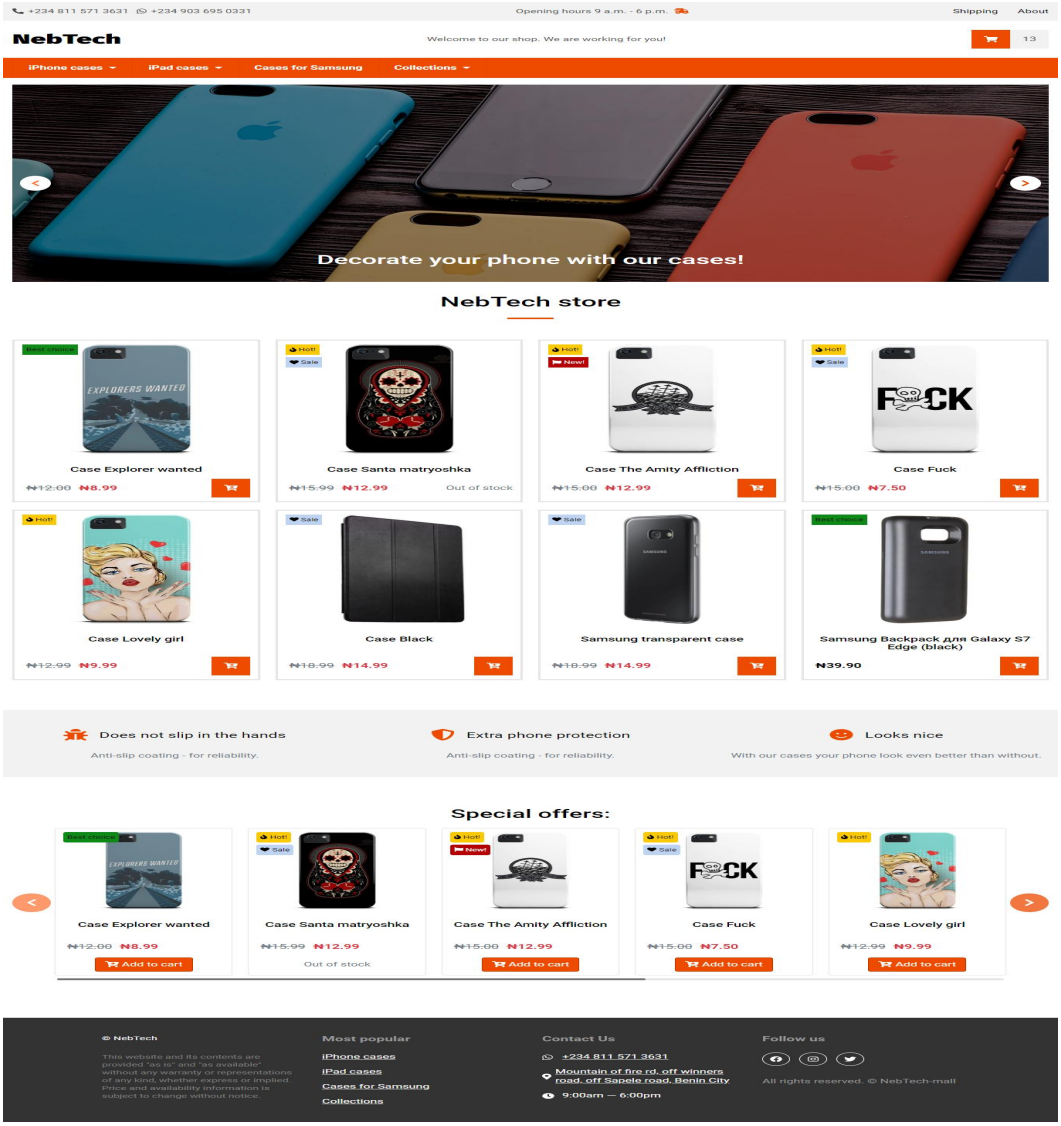

Plate 4.1 Overview of the E-commerce Homepage

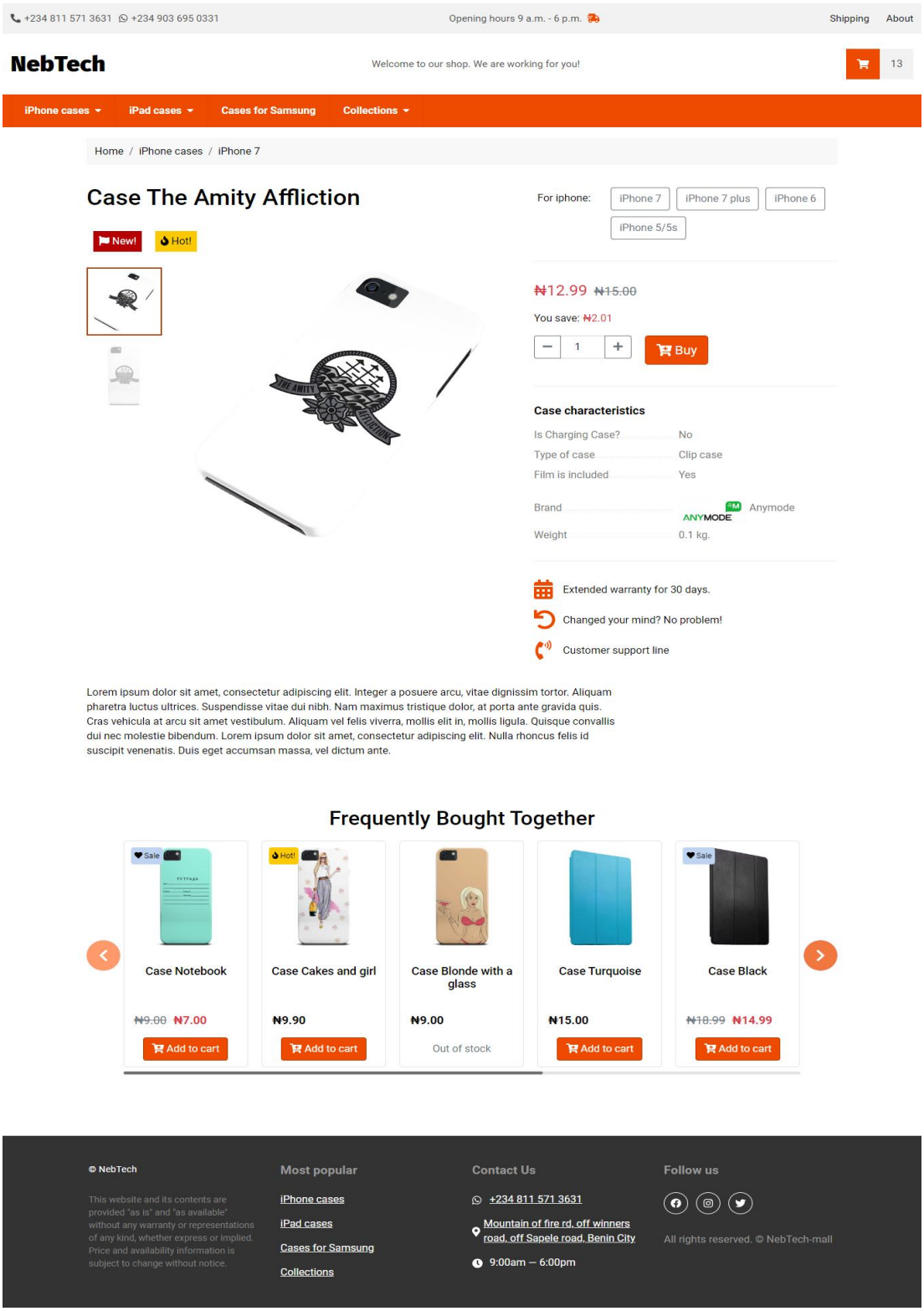

Plate 4.2 Overview of the E-commerce Product Page

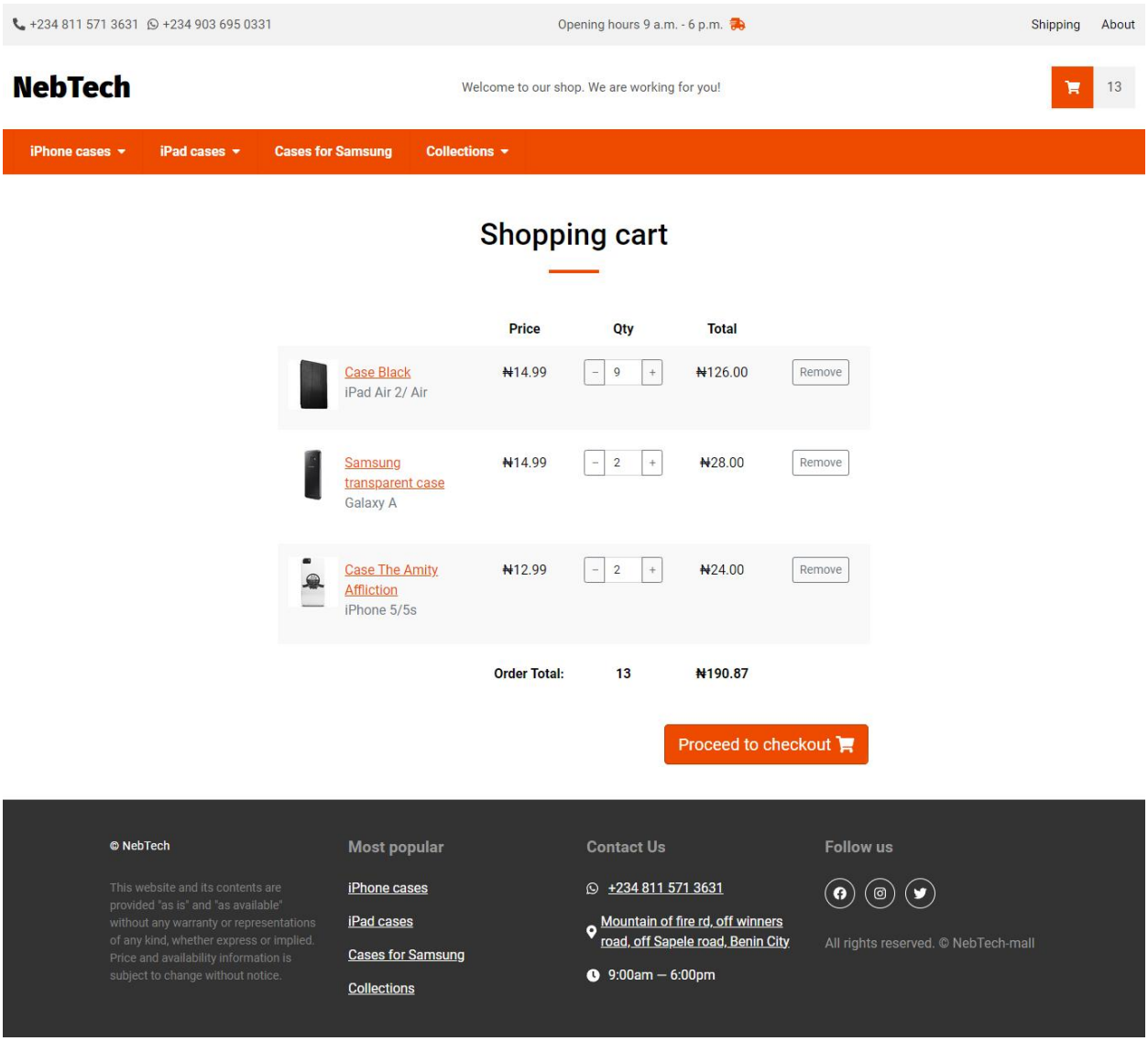

Plate 4.3 Overview of the E-commerce Cart Page

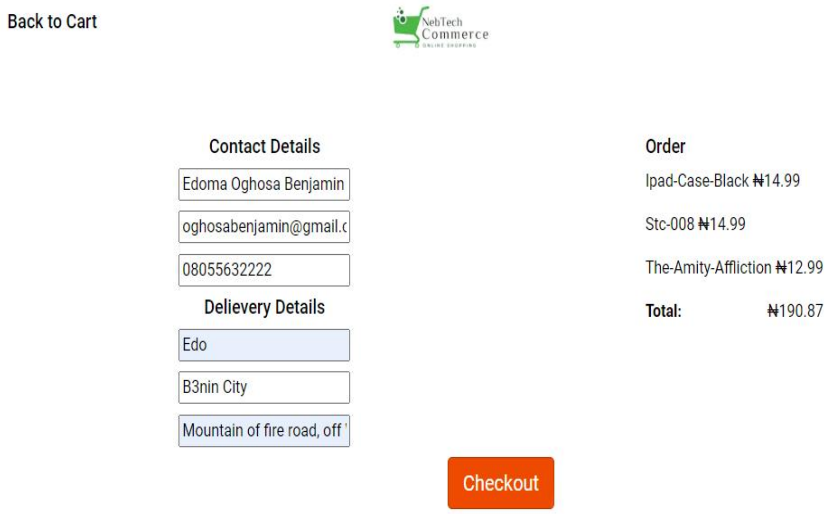

Plate 4.4 Overview of the E-commerce Checkout Page

4.3 Resource Management Analysis

Resource management is an important aspect of IT infrastructure and application deployment, and it plays a significant role in determining the efficiency, scalability, and cost-effectiveness of a system. Resource management includes the allocation, monitoring, optimization, and utilization of various computing resources, including CPU, memory, storage, network bandwidth, and more. In this scenario, resource management varied significantly between the serverless and non-serverless deployment methods,

4.3.1 Serverless Resource Management:

In the serverless computing method, resource management was abstracted and largely handled by the cloud provider. Some key aspects of resource management in serverless:

Automatic Scaling: The Serverless platform automatically allocated and deallocated resources based on its incoming workload. This means that as the number of requests or events increased, the platform automatically scaled out by providing additional instances to handle the load.

Conversely, it scaled in when the load decreased, terminating any unnecessary instances to save resources and costs.

No Server Management: There was no need to manage the server personally as this task was taken care of by the serverless platform.

Limited Control: Although the serverless computing method offered simplicity, it limited the control I had over low-level resource management. As I had to rely on the cloud provider to make optimization decisions, of which some did not align perfectly with my application's specific requirements.

4.3.2 Non-Serverless Resource Management:

In traditional non-serverless deployments, resource management is more hands-on and typically involves manual configuration and maintenance:

Manual Scaling: As opposed to serverless computing, In non-serverless environments, we are responsible for provisioning and scaling resources based on expected or observed workloads. This might involve setting up load balancers, auto-scaling groups, and monitoring systems to adjust resource allocation.

Full Control: Unlike serverless computing, we have full control over resource allocation, allowing us to optimize resources for cost, performance, and compliance. However, this control comes with increased operational complexity.

Server Management: There is a need to manage the server personally. This would involve ongoing maintenance, including security patching, scaling adjustments, and system updates.

4.4 Development and Deployment Speed Analysis

Development and deployment speed is another crucial factor in serverless and non-serverless deployment analysis, and it significantly impacts a project's time-to-market, agility, and overall

success. I analysed how both serverless and non-serverless deployment models affect development and deployment speed

4.4.1 Serverless Development and Deployment Speed

Development in Serverless was rapid because I was able to focus primarily on writing code without the need to manage infrastructure. This abstraction reduced the time spent on setting up servers, configuring networking, and handling operating system updates. The deployment was rapid also as it took only a few minutes and the serverless platform provided continuous deployment, that is automatic deployment when there are changes to the source code. This was done swiftly with minimal downtime.

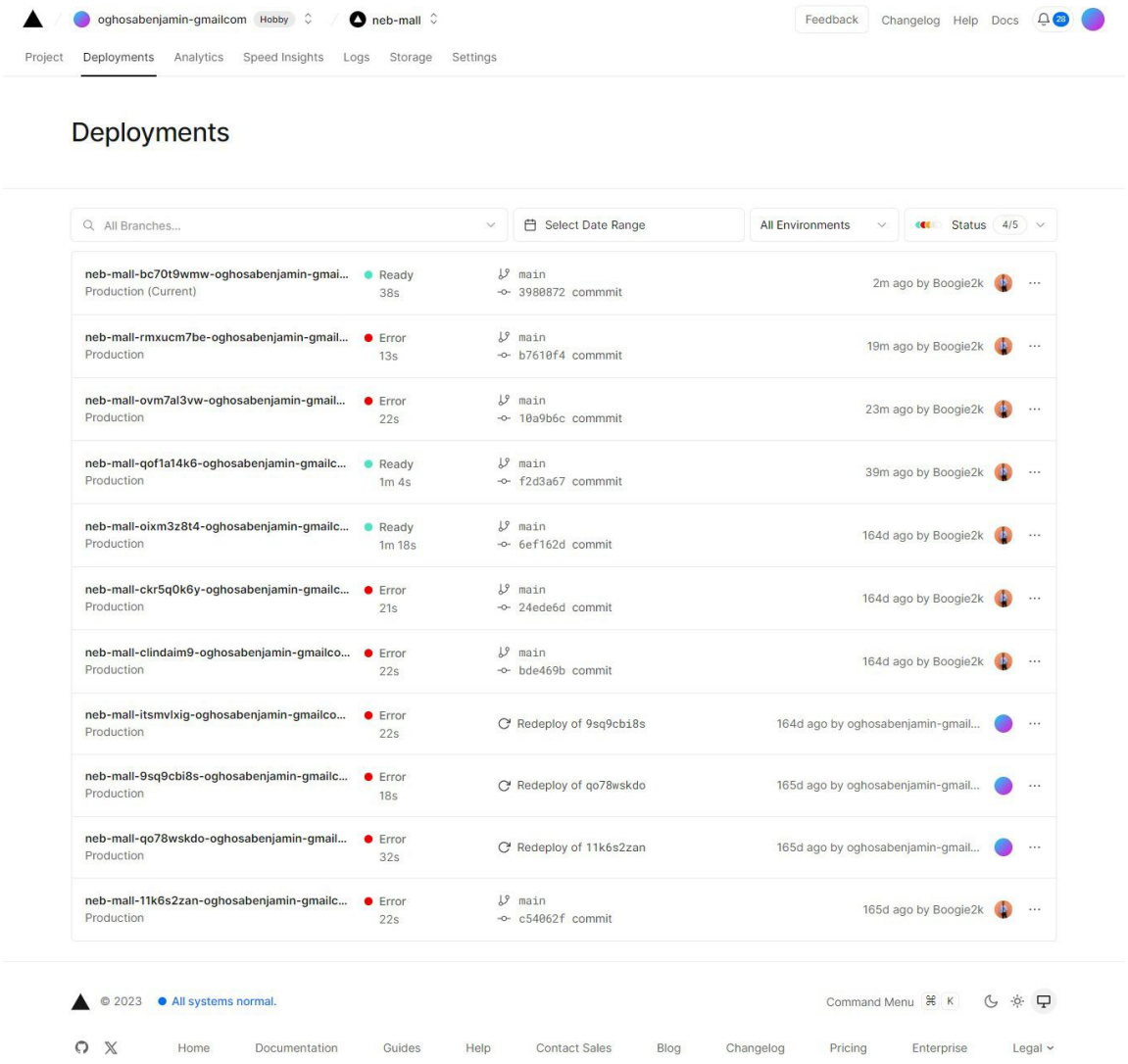

Plate 4.5 Snapshot of the cloud platform showing all the active deployments

4.4.2 Non-Serverless Development and Deployment Speed

In non-serverless deployments, developers would need to spend time setting up and configuring servers, virtual machines, and networking components. This would reduce the development speed of the developer. While cloud providers offer managed services, traditional deployments rely more on manually configuring and maintaining third-party components, increasing development time. Deploying applications in traditional environments involves complex processes, such as creating deployment scripts, managing dependencies, and ensuring

compatibility with the underlying infrastructure, this makes deployment more time-consuming and complex.

4.5 Performance Analysis

I evaluated various factors that impact the speed, efficiency, and responsiveness of the application in the context of serverless and non-serverless architecture. Here's an explanation of how performance considerations differed between these two deployment approaches:

4.5.1 Serverless Performance

The Serverless platform provided automatic scalability, dynamic resource allocation, and event-driven processing, allowing for efficient and responsive performance for the application. This model excelled in handling sudden traffic spikes by rapidly spinning up instances and providing low-latency responses to events. Although cold starts introduced slight latency for initial requests the pay-as-you-go cost model aligns with usage, making it cost-effective.

4.5.2 Non-Serverless Performance

In non-serverless deployments, due to the precise control over underlying resources, performance is consistent and predictable without cold start issues. Resource allocation and configuration are tailored to the application's needs, allowing for optimized performance in applications with stringent demands. With no cold starts and the ability to use specialized hardware or fine-tune environments, traditional deployments excel in applications requiring specific hardware resources or long-running processes. However, this control comes at the cost of increased complexity, manual scaling, and a less flexible cost structure, making it more suitable for applications with stable and well-defined resource requirements.

4.6 Security Analysis

Security is a critical consideration in both serverless and non-serverless deployments, the approach to security differs based on the deployment model. I compared both approaches and here's an explanation of security considerations for both.

4.6.1 Serverless Security

One big security advantage of the serverless architecture was its reliance on a reputable cloud provider, which typically furnished robust managed security features, including access controls, encryption, and automatic updates. It simplified least-privilege access with fine-grained permissions, employed strong isolation mechanisms, and integrated identity and access management services for authentication and authorization. Furthermore, the serverless platform also included monitoring and logging tools which facilitated the detection of security incidents. However, all this convenience came with the trade-off of giving up a significant portion of infrastructure security responsibility to the cloud provider, which raised trust and shared resource concerns, as misconfigurations or vulnerabilities in the provider's environment could pose risks to the application.

4.6.2 Non-Serverless Security

In contrast, non-serverless deployments would grant greater control and customization over security measures, which would enable proactive vulnerability management with patching and updates, the implementation of network-level security controls like firewalls, and fine-tuned access controls. This flexibility would extend to data encryption, compliance, and auditing, which would allow tailored solutions to meet specific requirements. However, the inherent control would also entail increased responsibility for security configuration and maintenance, making misconfigurations, and vulnerabilities more likely if not managed meticulously. The responsibility would be on the developer to design and implement a comprehensive security strategy, from access controls to incident response, which could be resource-intensive and necessitates expertise in various security domains.

4.7 Complexity Analysis

Complexity in the context of serverless and non-serverless deployment models refers to the level of intricacy and difficulty involved in designing, developing, deploying, and managing applications within each approach. Here's an explanation of the complexity associated with both deployment models:

4.7.1 Serverless Complexity

In serverless deployment, the complexity was reduced thanks to the cloud platform abstracting away server management, which simplified resource provisioning and enabled automatic scaling based on demand. However, it introduced intricacy in adopting an event-driven architecture, where the application was composed of small, stateless functions triggered by events, which required me to rethink application design and grapple with the challenges of cold starts and vendor-specific services integration. Monitoring and debugging also became complex due to the distributed nature of functions, which necessitated specialized tools and strategies, all of which offered the advantage of a simplified infrastructure management experience.

4.7.2 Non-Serverless complexity

In non-serverless deployment, complexity would primarily stem from the need to manage servers, virtual machines, and various infrastructure components. This would involve intricate tasks such as provisioning, configuring, and scaling resources manually, which would demand expertise in infrastructure management. Optimizing resource allocation, configuring middleware components, and ensuring high availability can add layers of complexity. Additionally, developing deployment pipelines and orchestrating different stages, including building, testing, and deploying, could be intricate and would require careful coordination. While non-serverless deployment would offer greater control, it would require meticulous configuration, which could be error-prone and time-consuming, this would make it necessary to address complexities related to infrastructure and resource management.

4.8 Monitoring and Debugging Analysis

Monitoring and debugging are critical aspects of both serverless and non-serverless deployments, but they differ in terms of the tools, techniques, and challenges involved. Here's an explanation of monitoring and debugging in the context of serverless and non-serverless deployments:

4.8.1 Monitoring in Serverless Deployment

Granular Metrics: The cloud platform(Vercel) provided granular metrics related to function execution, such as invocation counts, execution duration, and error rates.

Auto-scaling Insights: The Serverless platform automatically scaled the resources based on their demand. The Monitoring tools provided by the cloud platform showed how the functions scaled and helped me understand when and why auto-scaling occurs.

Event Tracing: The cloud platform provided distributed tracing tools which helped trace requests as they flowed through multiple serverless functions and services, aiding in pinpointing performance bottlenecks.

Error Tracking: The Serverless monitoring tools provided by the cloud platform aided in capturing and reporting errors, which helped to identify and address issues in the source code. This includes stack traces and error messages.

Cost Monitoring: Since the serverless platform charged based on execution, monitoring tools also provided insights into my costs, which helped me optimize my functions efficiently.

Real-time Alerts: The Monitoring solutions allowed for services like setting up alerts based on predefined thresholds for metrics like latency or error rates. This enabled proactive responses to issues

4.8.2 Debugging in Serverless Deployment

Local Debugging: The cloud platform offered local development and debugging tools, which allowed the functions of the application to be tested locally before deployment.

Remote Debugging: Remote debugging sessions were set up for issues that occurred in the cloud environment.

Logging: Logs are streamed to a centralized service for real-time analysis, this aided in debugging.

Code Versioning: The cloud platform provided proper version control for the application functions, so each time an issue arose after deployment, it was easy would roll back to a previous version for debugging purposes.

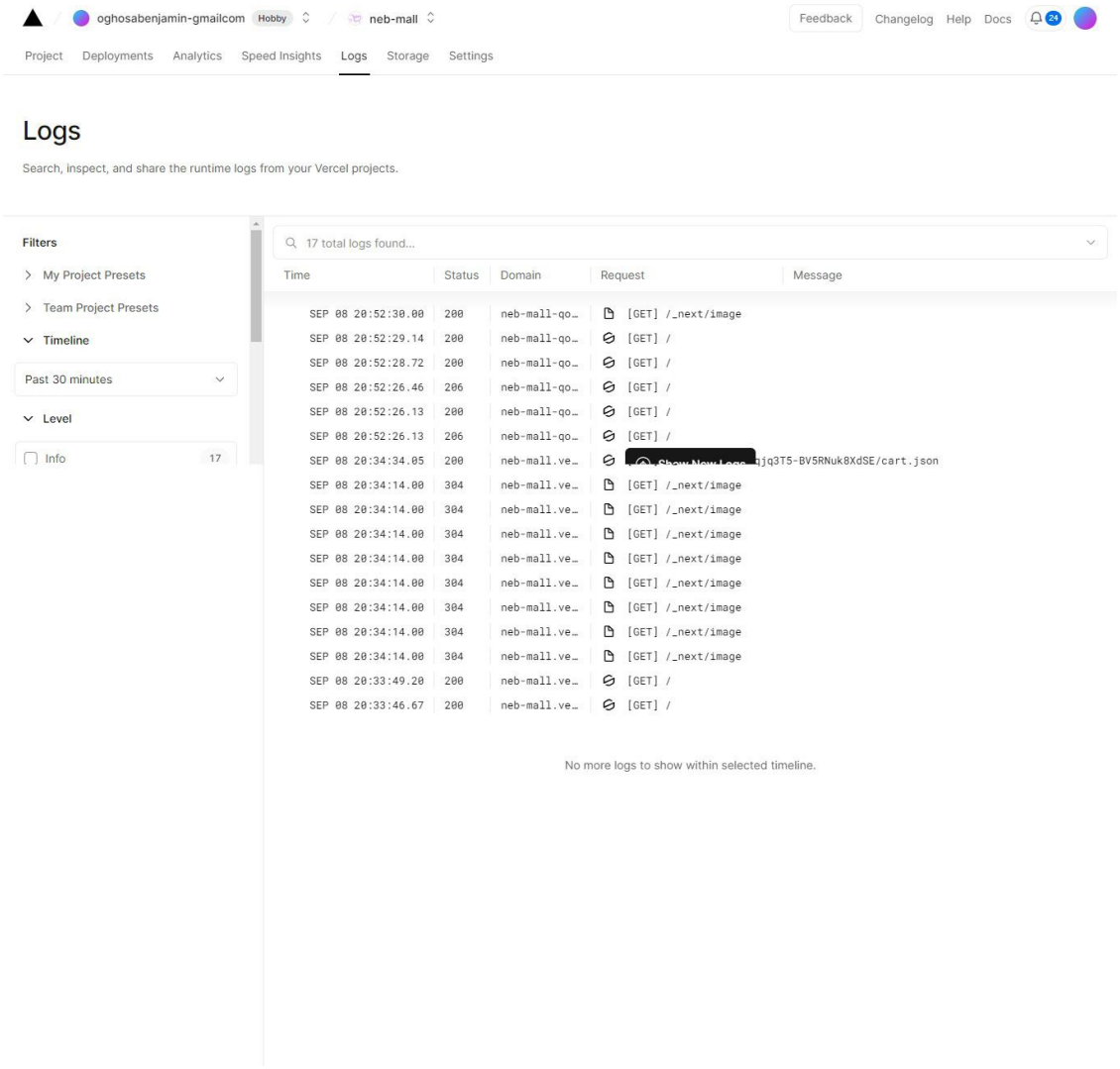

Plate 4.6 Snapshot of the cloud platform log showing all logs, which includes event changes, error messages, and successful requests.

4.8.3 Monitoring in Non-Serverless Deployment

Infrastructure Metrics: Non-serverless deployments would require monitoring at the infrastructure level, which would include servers, databases, load balancers, and network performance.

Application-Level Metrics: Besides infrastructure, there would be a need to monitor application-specific metrics such as database query performance, API response times, and application server resource utilization.

Log Aggregation: Centralized log aggregation tools like ELK Stack (Elasticsearch, Logstash, and Kibana) or Splunk could help consolidate logs from various components for analysis.

Custom Monitoring Solutions: There would be a need to set up custom monitoring scripts or agents to collect metrics from various components that are not automatically provided by cloud providers.

Alerting and Notification: Non-serverless deployments would often involve setting up custom alerting based on infrastructure and application metrics using tools like Prometheus or Nagios.

4.8.4 Debugging in Non-Serverless Deployment

Remote Debugging: Developers would need to SSH into servers or remote debugging tools to investigate issues in the production environment.

Logs and Stack: Traces: Logs are essential for debugging in non-serverless deployments. Developers rely on log files and stack traces to identify the source of errors.

Code Profiling: Profiling tools could be used to analyze the performance of specific functions or sections of code, this would help to pinpoint performance bottlenecks.

Code Instrumentation: Developers would need to add instrumentation to the codebase to collect additional data for debugging purposes, such as request/response tracing.

4.9 Cost Model Analysis

The "Cost Model" refers to how expenses are incurred and managed when deploying and running applications in the context of serverless and non-serverless (traditional) deployment models.

4.9.1 Serverless Deployment Cost Model

The serverless architecture followed a cost model characterized by pay-per-use precision, the I billing was solely based on actual function executions and resource utilization during each invocation, typically down to the millisecond level. This granular billing eliminated idle costs, which ensured resources consumed during execution were paid for, making it exceptionally cost-effective for variable workloads with sporadic traffic patterns. The Serverless platform automatically scaled to meet demand, mitigating the need for manual provisioning and enabling precise cost predictability.

4.9.2 Non-Serverless Deployment Cost Model

Non-serverless deployment models involve a fundamentally different cost model which is centred on fixed infrastructure costs, here the developer incurs expenses for servers, virtual machines, and related resources irrespective of their active use. This model would necessitate upfront resource provisioning, which might lead to either over-provisioning or under-provisioning and often entails scaling costs when adapting to variable workloads. Maintenance, software licensing, and data transfer expenses further contribute to the cost structure, while long-term commitments may be necessary to reduce costs but can limit flexibility. Non-serverless environments are best suited for applications with predictable workloads and established resource needs, providing more control but at the expense of increased financial and operational complexity.

4.10 Vendor lock-in Analysis

Vendor lock-in refers to the situation where a particular cloud provider's proprietary services, tools, or technologies make it challenging for you to migrate your application or data to another cloud provider or on-premises infrastructure. This concept applies to both serverless and non-serverless deployments, but the degree of lock-in can vary between these two approaches.

4.10.1 Serverless Deployment Vendor Lock-In

In serverless deployments, vendor lock-in is substantial due to the tightly coupled nature of applications with cloud providers' proprietary serverless platforms. This lock-in results from the reliance on specialized services, custom event triggers, and cloud-specific runtimes, making it difficult to transition to an alternative provider without substantial code refactoring. Additionally, the integration with cloud-native services, unique pricing models, and provider-specific monitoring tools further entrench the dependency on the chosen cloud vendor. These factors collectively pose a significant challenge to achieving portability and can impact long-term flexibility and cost control for serverless applications.

4.10.2 Non-Serverless Deployment Vendor Lock-In

In non-serverless or traditional deployments, vendor lock-in is somewhat less pronounced compared to serverless, primarily because users have greater control over infrastructure and software components. However, lock-in can still occur, primarily when relying on cloud-specific databases, APIs, or management tools. Transitioning to a different provider may entail adjustments to virtualization or containerization technologies, complex data migration processes, and adaptations to service integrations. While mitigation strategies like adopting cloud-agnostic tools and infrastructure as code can help alleviate some of these concerns, non-serverless deployments still necessitate careful consideration of vendor-specific dependencies and potential challenges associated with migration.

4.11 Latency Analysis

Latency refers to the time delay experienced in a system when data or requests are sent from a source to a destination and then a response is received. In the context of serverless and non-serverless deployments, latency can be impacted by the architecture and infrastructure choices you make. Here's an explanation of latency in terms of both serverless and non-serverless deployments.

4.11.1 Serverless Latency

The serverless architecture automatically provisioned resources in response to incoming requests or events, this would sometimes introduce initial latency which is also known as "cold starts", this would sometimes occur when the functions needed to initialize. However, once resources are provisioned, the serverless functions would execute with low latency and high efficiency due to their ability to handle requests or events in parallel. Moreover, the serverless architecture leveraged edge computing, which reduced latency by deploying resources closer to end-users. Automatic scaling ensured that resources were available during traffic spikes, this maintained low latency overall. Despite cold start delays, the serverless architecture excelled in handling individual requests swiftly, especially when it was deployed at edge locations, this provided a balance between initial latency and efficient response times.

4.11.2 Non-Serverless Latency

In non-serverless deployments, the developer has more control over the underlying infrastructure, allowing him to optimize for lower latency. By managing resources manually or through auto-scaling, the developer can keep servers "warm," reducing initial latency as they are ready to process requests immediately. Custom network configurations, strategic placement of servers, and the use of CDNs could further minimize latency. Fine-tuning the server environment, optimizing database queries, and maintaining persistent connections to services also contribute to reducing overhead and latency. Non-serverless architectures also provide flexibility and control, which would enable the reduction of latency through various optimizations, if minimizing

latency is a critical requirement in the application, then non-serverless architecture would be suitable for it.

4.12 Discussion

In this chapter, various aspects of serverless and non-serverless hosting infrastructures were analyzed and compared. The study covered resource management, development and deployment speed, performance, security, complexity, monitoring and debugging, cost models, vendor lock-in, and latency.

In terms of resource management, the serverless approach offered automatic scaling and abstracted server management, while the non-serverless approach required manual resource provisioning and server management, providing greater control but also increasing complexity.

Development and deployment speed favoured serverless, as it allowed rapid development without infrastructure management and offered automatic deployment, reducing downtime. The Non-serverless architecture required developers to configure servers and components, making development slower.

Performance analysis showed that serverless architectures excelled in handling traffic spikes with automatic scaling, but had slight initial latency due to cold starts. Non-serverless deployments provided consistent and predictable performance but required manual optimization.

Regarding security, serverless architectures benefited from robust cloud provider security features but required relinquishing some control. Non-serverless deployments allowed for customization but required meticulous security configuration and maintenance.

Complexity varied, with serverless simplifying infrastructure management but introducing complexity in adopting an event-driven architecture. Non-serverless deployments required expertise in infrastructure management but offered more control.

Monitoring and debugging tools differed significantly between the two approaches, with serverless offering granular metrics and cloud-native tools, while non-serverless required custom monitoring scripts and debugging through logs and remote access.

Cost models favoured serverless for variable workloads, with granular pay-per-use billing, while non-serverless involved fixed infrastructure costs and complexity in resource provisioning.

Vendor lock-in was substantial in serverless due to proprietary services, whereas non-serverless deployments had some dependencies but offered more flexibility with careful planning.

Finally, latency was impacted by cold starts in serverless but was optimized for low latency with automatic scaling and edge computing. Non-serverless deployments allowed for low latency optimization through manual resource management.

This analysis highlighted the trade-offs between serverless and non-serverless hosting infrastructures, with each approach offering unique advantages and challenges in resource management, development speed, performance, security, complexity, monitoring, cost, vendor lock-in, and latency. The choice between the two depends on specific project requirements and priorities.

In the following chapter, I will conclude this research and suggest a few ideas on how the work can be improved.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.1 CONCLUSION

Serverless and Non-serverless deployment is of great importance today, especially in the field of e-commerce and internet-related activities/applications. With it, applications/services are made available to end-users across the globe. This research has provided valuable insights into the dynamic landscape of modern web hosting solutions which would guide individuals, developers and even organisations toward informed decisions that can lead to improved performance, cost-efficiency, and customer satisfaction in the competitive online marketplace. This study explored the advantages and drawbacks of utilizing serverless and non-serverless hosting infrastructures, coupled with Software as a Service (SaaS) implementation, within the context of e-commerce websites. From this research, It is evident that both serverless and non-serverless hosting approaches have their unique merits and limitations. With Serverless infrastructure, its auto-scaling capabilities and cost-effectiveness present an appealing option for businesses looking to optimize resource utilization and minimize operational overhead. On the other hand, non-serverless infrastructures offer more control and customization but require greater management and monitoring efforts.

In this research, four key objectives were pursued. Firstly, the study focused on obtaining the deployment steps for serverless infrastructure, delving into the intricate process of deploying applications within this transformative hosting paradigm. This exploration included an examination of various cloud service providers such as AWS Lambda, Azure Functions, and Vercel, emphasizing the importance of security and authentication mechanisms to ensure data integrity and confidentiality within serverless environments. Secondly, the research aimed to document the deployment steps for non-serverless infrastructure, emphasizing the need for meticulous provisioning of physical or virtual servers, installation, and configuration of essential components, load balancing, redundancy, and scalability considerations to cater to the specific needs of traditional digital applications.

The third objective was a critical analysis of the performance, cost implications, and various factors affecting both serverless and non-serverless hosting infrastructure. Through empirical evaluation, the research illuminated the nuanced dynamics of modern web hosting solutions. While serverless architectures offer cost optimization potential, particularly with fluctuating demand, however, they may face performance bottlenecks under extreme loads. Non-serverless infrastructures provided greater control over performance but often involved higher upfront and operational costs. These findings enable organizations to make informed decisions tailored to their unique requirements and financial constraints.

Lastly, in Chapter 3, a serverless-based application was developed, specifically an e-commerce website deployed on Vercel, providing a practical demonstration of serverless architecture. This served as a basis for comparing the application's performance and cost implications with a non-serverless deployment approach. The insights derived from this research can assist developers, architects, and cloud service providers in making informed decisions about hosting strategies, ultimately optimizing the delivery of SaaS solutions with improved efficiency, scalability, and cost-effectiveness

.

5.2 RECOMMENDATION

For further research, non-serverless deployment with the use of virtual machines should be highlighted more. On the analysis of serverless deployment more serverless cloud platforms should be analysed, this will help developers not only have a better understanding of serverless deployments but also have a better understanding of how cloud platforms work in general.

REFERENCES

Ivan, C. & Dadarlat, V. (2019). Serverless Computing: An Investigation of Deployment Environments for Web APIs. ResearchGate. Retrieved from https://www.researchgate.net/publication/334015883_Serverless_Computing_An_Investigation_of_Deployment_Environments_for_Web_APIs.

Hassan, H. B., Barakat, S. A., & Sarhan, Q. I. (2021). Survey on serverless computing..

Adzic, G., & Chatley, R. (2017). Serverless computing: Economic and architectural impact.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017). Serverless Computing: Current Trends and Open Problems.

Truong, D. (2010, January). How cloud computing enhances competitive advantages: A research model for small businesses.

Haroon-Sulyman, S. (2014). Client-Server Model.

Kinsta. (2022). What Is GitHub? A Beginner's Introduction to GitHub.

Reverus (2022). Physical Server vs. Virtual Server: Which Is Right for You?

Roman Kolodiy (2022). Serverless Deployment: How It Works

Vitaly Makhov (2021). Server vs. Serverless: Benefits and Downsides

CalcProgrammer1(2022). Set Up Your Very Own Web Server

Cisco Router and Switch Forensics (2009). Virtual Network Computing.

Wesley Chai (2022). HTTP (Hypertext Transfer Protocol)

What is FTP (File Transfer Protocol)? (2022). Sean Michael Kerner and John Burke

APPENDIX

Software source code

The complete source code for the e-commerce website can be located at:
https://github.com/Boogie2k/Neb-Mall

Source code for the homepage

```
import {IProduct} from 'boundless-api-client';

import {GetServerSideProps, InferGetServerSidePropsType} from 'next';

import ProductsList from '../components/ProductsList';

import MainLayout from '../layouts/Main';

import {apiClient} from '../lib/api';

import {makeAllMenus} from '../lib/menu';

// import VerticalMenu from '../components/VerticalMenu';

import {IMenuItem} from '../@types/components';

import SwiperSlider from '../components/SwiperSlider';

import mobileSlider1Img from '../assets/mobile-slider-1.png';

import mobileSlider2Img from '../assets/mobile-slider-2.png';

// import CoverTextInCenter from '../components/CoverTextInCenter';

// import bgImg from '../assets/cover-bg.jpeg';

// import bgPortraitImg from '../assets/cover-bg-portrait.jpg';

import ProductsSliderByQuery from '../components/ProductsSliderByQuery';

import TextWithIcons from '../components/TextWithIcons';

import {faBug} from '@fortawesome/free-solid-svg-icons/faBug';
```

```
import {faShieldAlt} from '@fortawesome/free-solid-svg-icons/faShieldAlt';
import {faSmile} from '@fortawesome/free-solid-svg-icons/faSmile';
import {FontAwesomeIcon} from '@fortawesome/react-fontawesome';
export default function IndexPage({products, mainMenu, footerMenu}: InferGetServerSidePropsType<typeof getServerSideProps>) {
    return (
        <MainLayout mainMenu={mainMenu} footerMenu={footerMenu}>
            <div className='container-xxl'>
                <MainPageSlider />
                <h1 className='page-heading page-heading_h1  page-heading_m-h1'>NebTech store</h1>
                {/*<Image src={logoImg} className='page-heading page-heading_h1  page-heading_m-h1'/>*/}
                <ProductsList
                    products={products}
                    className={'page-block'}
                    itemClassName={'products__item_4-in-row'}
                />
            </div>
            <TextWithIcons
                columns={[
                    {
```

```
                                    icon: <FontAwesomeIcon icon={faBug} className={'text-with-icons__icon'} />,
                                    title: 'Does not slip in the hands',
                                    comment: 'Anti-slip coating - for reliability.'
                                },
                                {
                                    icon: <FontAwesomeIcon icon={faShieldAlt} className={'text-with-icons__icon'} />,
                                    title: 'Extra phone protection',
                                    comment: 'Anti-slip coating - for reliability.'
                                },
                                {
                                    icon: <FontAwesomeIcon icon={faSmile} className={'text-with-icons__icon'} />,
                                    title: 'Looks nice',
                                    comment: 'With our cases your phone look even better than without.'
                                },
                            ]}
                            fullWidth={true}
                            className={'page-block'}
                        />
```

```
<div className='container-xxl'>
    <ProductsSliderByQuery
        query={{collection: ['main-page'], sort: 'in_collection'}}
        title={'Special offers:'}
        wrapperClassName='page-block'
    />
    {/* <div className={'page-block'}>
        <h2 className={'text-center mb-4'}>Our customers love us:</h2>
        <Reviews
            reviews={[
                {
                    image: <img src={reviewWoman1.src} className={'reviews__img'} />,
                    title: 'Amanda',
                    jobTitle: 'CEO reseller corp',
                    comment: 'I like working with the wholesales team. We are thankful for your great service!'
                },
                {
                    image: <img src={reviewMan1.src} className={'reviews__img'} />,
```

```
                                                    title: 'Jack',
                                                    jobTitle: 'Frequent buyer',
                                                    comment: 'I like the quality and the quick shipping.'
                                                },
                                                {
                                                    image: <img src={reviewMan2.src} className={'reviews__img'} />,
                                                    title: 'Dave',
                                                    jobTitle: 'Founder at Startup',
                                                    comment: 'I love how the things are going!            },]}     />
                                    </div> */}</div>
                    </MainLayout>);}
export const getServerSideProps: GetServerSideProps<IIndexPageProps> = async () => {
        const categoryTree = await apiClient.catalog.getCategoryTree({menu: 'category'});
        const {products} = await apiClient.catalog.getProducts({collection: ['main-page'], sort: 'in_collection'});
        const menus = makeAllMenus({categoryTree});
        return {
                props: {
                        products,
```

```
                    ...menus        }};};
interface IIndexPageProps {
        products: IProduct[];
        mainMenu: IMenuItem[];
        footerMenu: IMenuItem[];
}

function        MainPageSlider() {
        const slides = [
                {
                        'img': mobileSlider1Img.src,
                        'link': '',
                        'caption': 'Decorate your phone with our cases!',
                        'captionPosition': 'bottom',
                        'useFilling': true,
                        'fillingColor': '#000000',
                        'fillingOpacity': 0.40
                },
                {
                        'img': mobileSlider2Img.src,
                        'link': '',
```

```
                'caption': 'Pray not for easy lives, pray to be stronger men.',
                'captionPosition': 'bottom',
                'useFilling': true,
                'fillingColor': '#000000',
                'fillingOpacity': 0.4
            }
        ];
        return (
            <SwiperSlider
                showPrevNext
                // pagination='progressbar'
                size={'large'}
                slides={slides}
                className={'mb-4'}/>);}
```

Product Page

```
import {useEffect, useMemo, useState} from 'react';
import {ICategoryFlatItem, IGetProductsParams, IProductItem} from 'boundless-api-client';
import {GetStaticPaths, GetStaticProps, InferGetStaticPropsType} from 'next';
import MainLayout from '../../layouts/Main';
import {apiClient} from '../../lib/api';
import {useRouter} from 'next/router';
```

```
import BreadCrumbs from '../../components/BreadCrumbs';

import ProductImages from '../../components/product/Images';

import qs, {ParsedQs} from 'qs';

import MetaSchemaOrg from '../../components/product/MetaSchemaOrg';

import {getProductMetaData} from '../../lib/meta';

import ProductLabels from '../../components/product/Labels';

import ProductVariantAndBuy from '../../components/product/VariantAndBuy';

import ProductCharacteristics from '../../components/product/Characteristics';

import {makeAllMenus} from '../../lib/menu';

import {makeBreadCrumbsFromCats} from '../../lib/breadcrumbs';

import ProductShipping from '../../components/product/Shipping';

import {IMenuItem} from '../../@types/components';

import ProductsSliderByQuery from '../../components/ProductsSliderByQuery';

export default function ProductPage({data: {product, categoryParents, mainMenu, footerMenu}}:
InferGetStaticPropsType<typeof getStaticProps>) {

    const [resolvedParents, setResolvedParents] = useState(categoryParents);

    const router = useRouter();

    const query = useMemo<ParsedQs>(() => qs.parse(router.asPath.split('?')[1] || ''),
[router.asPath]);

    const {category, ...restQuery} = query;

    const similarQuery = useMemo(() => ({cross_sell_category: 'similar', cross_sell_product:
product.product_id}), [product]);
```

```
const relatedQuery = useMemo(() => ({cross_sell_category: 'related', cross_sell_product: product.product_id}), [product]);

const fetchParents = async (categoryId: number) =>

    setResolvedParents(await apiClient.catalog.getCategoryParents(categoryId));

useEffect(() => {

    const categoryId = category ? parseInt(category as string) : null;

    if (!categoryId) return;

    const notDefaultCat = product.categoryRels.some(cat => (cat.is_default !== true && cat.category_id === categoryId));

    if (notDefaultCat) {

        fetchParents(categoryId);

    }

}, [category, product]);

const breadcrumbItems = useMemo(() => {

    return makeBreadCrumbsFromCats(resolvedParents || [], ({category_id}) => {

        if (resolvedParents?.length && category_id === resolvedParents[0].category_id) {

            return {

                queryParams: restQuery

            };

        }

        return {};
```

```
            });
        }, [resolvedParents, query]); //eslint-disable-line
        return (
            <MainLayout
                footerMenu={footerMenu}
                mainMenu={mainMenu}
                metaData={getProductMetaData(product)}
                title={product.seo.title}
            >
<div className={'container'}>
                        <BreadCrumbs items={breadcrumbItems} />
                        <div className='product-page' itemScope itemType='//schema.org/Product'>
                            <div className='row'>
                                <div className='col-md-7'>
                                    <h1 className='product-page__header mb-4' itemProp='name'>
                                        {product.text.title}
                                    </h1>
                                    <ProductLabels labels={product.labels} className={'mb-3'} />
                                    <ProductImages product={product} />
```

```
                                        </div>
                                        <div className='col-md-5'>
                                            <ProductVariantAndBuy product={product}
/>
<hr className='product-page__hr' />
        <ProductCharacteristics
        characteristics={product.nonVariantCharacteristics!}
                                                manufacturer={product.manufacturer}
                                                    size={product.props.size}
                                                />
                                                <hr className='product-page__hr' />
                                                <ProductShipping />
                                        </div>
                                    </div>
                                    {product.text.description && <article
itemProp='description'
                                            className={'product-page__description'}
                                            dangerouslySetInnerHTML={{__html:
product?.text.description}} />}
<MetaSchemaOrg product={product} parents={resolvedParents} />
                                </div>
                                <ProductsSliderByQuery
```

```
                    query={similarQuery as IGetProductsParams}
                    title='Similar products'
                    wrapperClassName='page-block'
                />
                <ProductsSliderByQuery
                    query={relatedQuery as IGetProductsParams}
                    title='Frequently Bought Together'
                    wrapperClassName='page-block'/>
            </div>
        </MainLayout>);}
export const getStaticPaths: GetStaticPaths = async () => {
    const {pagination, products} = await apiClient.catalog.getProducts({'per-page': 100});
    if (pagination.pageCount > 1) {
        for (let page = 2; page <= pagination.pageCount; page++) {
            const {products: newProducts} = await apiClient.catalog.getProducts({'per-page': 100, page});
            products.push(...newProducts);
        }
    }
    const paths = products.map(product => ({
        params: {
```

```
                slug: product.url_key || String(product.product_id)
            }
        }));
        return {
            paths,
            fallback: 'blocking'
        };
};
export const getStaticProps: GetStaticProps<IProductPageProps> = async ({params}) => {
    const {slug} = params || {};
    let data = null;
    try {
        data = await fetchData(slug as string);
    } catch (error: any) {
        if (error.response?.status === 404) {
            return {
                notFound: true
            };
        } else {
            throw error;
        }
```

```
        }
        if (data?.product?.text.url_key && data?.product?.text.url_key !== slug) {
                return {
redirect: {
destination: `/product/${data?.product?.text.url_key}`,
permanent: true,
                        }};
        }
        return {
                props: {data    }};};
const fetchData = async (slug: string) => {
        const product = await apiClient.catalog.getProduct(slug as string);
        const categoryId = product.categoryRels.find(cat => cat.is_default === true)?.category_id;
        let categoryParents = null;
        if (categoryId) {
                categoryParents = await apiClient.catalog.getCategoryParents(categoryId);
        }
        const categoryTree = await apiClient.catalog.getCategoryTree({menu: 'category'});
        const menus = makeAllMenus({categoryTree});
```

```
        return {
                product,
                categoryParents,
                ...menus
        };
};
interface IProductPageProps {
        data: IProductPageData;
}
interface IProductPageData {
        product: IProductItem;
        categoryParents: ICategoryFlatItem[] | null;
        mainMenu: IMenuItem[];
        footerMenu: IMenuItem[];
}
```

Cart Page

```
import {ICartItem} from 'boundless-api-client';
import {useEffect, useMemo, useState} from 'react';
import CartItems from '../components/cart/CartItems';
import {useAppDispatch} from '../hooks/redux';
import MainLayout from '../layouts/Main';
```

```
import {apiClient} from '../lib/api';
import {setCartTotal, TCartInited} from '../redux/reducers/cart';
import {addPromise} from '../redux/reducers/xhr';
import {useCart} from '../hooks/cart';
import {makeAllMenus} from '../lib/menu';
import {IMenuItem} from '../@types/components';
import {GetServerSideProps} from 'next';
import CartLoader from '../components/cart/CartLoader';
import Link from 'next/link';
import {calcTotal, calcTotalPrice} from '../lib/calculator';
export default function CartPage({mainMenu, footerMenu}: ICartPageProps) {
        const dispatch = useAppDispatch();
        const {id: cartId, cartInited} = useCart();
        const [items, setItems] = useState<ICartItem[]>([]);
        const [loading, setLoading] = useState(false);
        const getCartData = async (cartId: string) => {
                setLoading(true);
                const promise = apiClient.cart.getCartItems(cartId)
                        .then(({cart, items}) => {
                                setItems(items);
                                dispatch(setCartTotal(cart.total));
```

```
                })
                .catch((err) => console.error(err))
                .finally(() => setLoading(false));
        dispatch(addPromise(promise));
};
const total = useMemo(() => calcTotal(items.map(el => ({
        qty: el.qty,
        price: calcTotalPrice(el.itemPrice.final_price!, el.qty)
}))), [items]);
useEffect(() => {
        dispatch(setCartTotal({
                qty: total.qty,
                total: total.price
        }));
}, [total]); //eslint-disable-line
useEffect(() => {
        if (cartId) getCartData(cartId);
}, [cartId]); //eslint-disable-line
return (
        <MainLayout mainMenu={mainMenu} footerMenu={footerMenu} noIndex>
                <div className='container'>
```

```
                                <div className='cart-page row'>
        <div className='col-lg-8 offset-lg-2'>
                                                <h1    className='page-heading    page-heading_h1
page-heading_m-h1'>Shopping cart</h1>
                                                <div className='cart-page__content'>
{(loading || cartInited === TCartInited.processing)? <CartLoader />: items.length > 0?
        <div><CartItems items={items} setItems={setItems} total={total}/>


</div>
: <>
<p className='cart-page__warning'>
Your shopping cart is empty.
</p>
<p className='cart-page__warning'>
<Link href='/'>
<a className='btn btn-success'>Go shopping!</a>


        </Link></p></>
}</div></div></div>                    </div>
                </MainLayout >
        );
}
export const getServerSideProps: GetServerSideProps<ICartPageProps> = async () => {
```

```
    const categoryTree = await apiClient.catalog.getCategoryTree({menu: 'category'});
    const {mainMenu, footerMenu} = makeAllMenus({categoryTree});
    return {
        props: {
            mainMenu,
            footerMenu
        }
    };
};
interface ICartPageProps {
    mainMenu: IMenuItem[];
    footerMenu: IMenuItem[];
}
```

Checkout page

```
import {useCart} from '../../hooks/cart';
//import {StarterWrapper} from 'boundless-checkout-react';
import {ICartItem} from 'boundless-api-client';
import {useRouter} from 'next/router';
import {apiClient} from '../../lib/api';
import {setCartTotal} from '../../redux/reducers/cart';
//import Loader from '../../components/Loader';
```

```
import logoImg from '../../assets/new-logo.png';/* logo image */
import Head from 'next/head';
import {useEffect,  useState, useMemo} from 'react';
import {useAppDispatch} from '../../hooks/redux';
import {addPromise} from '../../redux/reducers/xhr';
import {calcTotal, calcTotalPrice} from '../../lib/calculator';
import checkStyles from '../../styles/Checkout.module.css';
import {usePaystackPayment} from 'react-paystack';
import Image from 'next/image';
export default function CheckoutPage() {
        const [name, setName] = useState('');
  const [email, setEmail] = useState('');
   const [num, setNum] = useState('');
    const [city, setCity] = useState('');
  const [state, setState] = useState('');
   const [address, setAddress] = useState('');
        const dispatch = useAppDispatch();
        const {id: cartId} = useCart();
        const [items, setItems] = useState<ICartItem[]>([]);
        const [loading, setLoading] = useState(false);
        const getCartData = async (cartId: string) => {
```

```
if(num){
        console.log(loading);
}
                setLoading(true);
                const promise = apiClient.cart.getCartItems(cartId)
                        .then(({cart, items}) => {
                                setItems(items);
                                dispatch(setCartTotal(cart.total));
                        })
                        .catch((err) => console.error(err))
                        .finally(() => setLoading(false));
        dispatch(addPromise(promise));
        };
        const total = useMemo(() => calcTotal(items.map(el => ({
                qty: el.qty,
                price: calcTotalPrice(el.itemPrice.final_price!, el.qty)
        }))), [items]);

        useEffect(() => {
                dispatch(setCartTotal({
                        qty: total.qty,
```

```
                    total: total.price
                }));
        }, [total]); //eslint-disable-line
        useEffect(() => {
                if (cartId) getCartData(cartId);
        }, [cartId]); //eslint-disable-line
        //console.log(items)
        //console.log(total)
//items.map(item=>{
        //console.log(item.vwItem.product.sku)})
        const b = total.price;
 const newString = b.replace('₦', '');
        const router = useRouter();
   const config = {
    reference: new Date().getTime().toString(),
    email: email,
    name: name,
    amount:+newString* 100, //Amount is in the country's lowest currency. E.g Kobo, so 20000 kobo = N200
    phoneNumber:num,
    publicKey: 'pk_test_d0725f67a2558608e3a5cb2d22d197d2eeb3cc5c',
```

```
  };
 const onSuccess = () => {
    // Implementation for whatever you want to do with reference and after success call.
   /*  alert(reference);
    alert('Order has been successful, You will receive an email few hours from now and your products will be delivered shortly ');
            router.push('/');
  };
  // you can call this function anything
  const onClose = () => {
    // implementation for  whatever you want to do when the Paystack dialog closed.
    console.log('closed');
  };
const initializePayment = usePaystackPayment(config);
 const concludeCheckout=()=>{
if(total){
    if(name&&email &&address&&num&&state&&city){
      initializePayment(onSuccess, onClose);}
    else {
   alert('please fill in all delivery and contact details');
    }}
```

```
        else{
                alert('Cart is empty');
        }
  };
        return (
                <>
        <Head>
                                <meta name='robots' content='noindex' />
                        </Head>
                         <div  style={{backgroundColor:'white'}}>
<div className={checkStyles.contain}>
        <div className={checkStyles.checkoutNav}>
                <h4                                                onClick={()=>{router.push('/cart');}}
className={checkStyles.checkoutBTC}>Back to Cart</h4>
                <Image onClick={()=>{router.push('/');}} src={logoImg} alt='img' width={200}
height={200}/>
        </div>
<div className={checkStyles.containRows}>
        <section>
<div  className={checkStyles.deliveryDetails}>
        <h4 className={checkStyles.checkoutInputHeaders}> Contact Details</h4>
        <input   className={checkStyles.input} type='text' placeholder='Name'   value={name}
```

```
        onChange={(e) => {
          setName(e.target.value);
        }}/>
          <input  className={checkStyles.input} type='Email' placeholder='email'value={email}
        onChange={(e) => {
          setEmail(e.target.value);
        }}  />
          <input  value={num}
        onChange={(e) => {
          setNum(e.target.value);
        }} className={checkStyles.input} type='text' placeholder='Phone number'/>
</div>
<div className={checkStyles.deliveryDetails}>
<h4  className={checkStyles.checkoutInputHeaders}>Delievery Details</h4>
          <input className={checkStyles.input} type='text' placeholder='state' value={state}
        onChange={(e) => {
          setState(e.target.value);
        }}/>
          <input className={checkStyles.input} type='Email' placeholder='City ' value={city}
        onChange={(e) => {
          setCity(e.target.value);
```

```
        }}/>
         <input className={checkStyles.input} type='text' placeholder='address' value={address}
        onChange={(e) => {
         setAddress(e.target.value);
        }}/>
</div>
</section>
<div className={checkStyles.orderDiv}>

<h4> Order</h4>
{
items&& items.map(item=>{
        return(
                <div>
        <p className={checkStyles.productsNames}>  {item.vwItem.product.sku+ '  ' } <span
style={{textAlign:'right'}}>{'₦'+ item.itemPrice.final_price}</span> </p>
</div>
);})}
<p style={{display:'flex', justifyContent:'space-between'}}> <span
style={{fontWeight:'bold'}}>Total:</span> {total.price}</p>
</div>
</div>
```

<button onClick={concludeCheckout} className={checkStyles.checkoutBtn+ ' btn btn-action btn-lg btn-anim'}>Checkout</button>

 </>);}