Building and Deploying a Data Reporting Web Application: Integrating Salesforce, Google Analytics, and Azure Storage with the PERN Stack Technology

Md Aminul Islam

Haaga-Helia University of Applied Sciences
Degree in Business Information Technology
Bachelor Thesis 2025

Abstract

Author
Md Aminul Islam

Degree
Bachelor of Business Administration

Report/Thesis Title

Building and Deploying a Data Reporting Web Application: Integrating Salesforce, Google Analytics, and Azure Storage with the PERN Stack Technology

Number of pages
67

This thesis focuses on the design, development, and deployment of a data reporting application which facilitates the collection of users data from Salesforce, Google Analytics, and Azure Storage using the PERN stack (PostgreSQL, Express.js, React, Node.js). The project addresses the inefficiencies of manual data reporting processes by introducing an automated system that can ensure scalability, accuracy, and real-time data insights. The implementation is commissioned by Mobimus Oy, operating under Osuria brand, which provides digital secretary services to property management companies.

The application uses APIs to fetch data from multiple sources and displaying it through an interactive dashboard developed with React. The backend, built using Node.js and Express, follows a modular architecture while PostgreSQL stores authentication data to ensure efficiency. The frontend is implemented using React components to provide dynamic user interactions and data visualizations, improving user experience. In terms of secure communication between frontend and backend, JWT is implemented as authentication approach.

By using Docker for containerization and Azure Container Apps and Container registry for cloud deployment, the project highlights modern deployment practices. The Dockerized application ensures consistent performance across development, and production environments while Azure Container Registry and Azure Container Apps provide Perfect Combination for Containerized Application Deployment.

The results demonstrate the practical advantages behind data automation, such as reduced manual workload, and real-time access to the information. The project contributes to a broader understanding of multi-sources REST APIs applied in application development. It highlights best practices in using containerization and cloud services to create scalable and efficient business applications.

Key words
Data Reporting, Cloud Deployment, PERN Stack, Salesforce Integration, Google Analytics, Azure Container Apps, REST API, React, Node, PostgreSQL, Express, Component.

Content

1 Introduction ... 4
1.1 Commissioning Organization ... 5
1.2 Background and Motivation ... 5
1.3 Problem Statement and Research Question ... 6
1.4 Research Methodology ... 7
1.5 Research Approach ... 7
1.6 Scope and Limitations of the Thesis ... 7
1.7 Structure of the Thesis ... 8
2 Theoretical Perspectives on Data Integration & Reporting ... 9
2.1 The Importance of Automated Data Reporting ... 9
2.2 Overview of Salesforce, Azure, and Google Analytics as Data Sources ... 9
2.3 Common Challenges in Manual Data Reporting Processes ... 10
2.4 The Role of Web Applications in Solving Data Reporting Challenges ... 10
2.5 Data Source Integration - Modular Architecture ... 10
2.6 Security Considerations and Justification for JWT ... 11
2.7 Testing Methodology and Analysis of Results ... 12
3 Technologies Used in the Project ... 13
3.1 Overview of the PERN Stack ... 13
3.2 PostgreSQL: Database for Storing Data ... 14
3.3 Express.js: Backend Framework for API Development ... 14
3.4 React: Frontend Framework for Building User Interfaces ... 15
3.5 Node.js: Runtime Environment for Backend Services ... 16
3.6 Docker: Containerization Technology ... 16
3.7 Azure Cloud Services ... 18
3.8 REST APIs: Connecting to Salesforce, Azure, and Google Analytics ... 18
4 Application Deployment Concepts ... 19
4.1 Understanding Cloud Deployment ... 19
4.2 The Role of Containerization in Modern Software Development ... 19
4.3 Scalability and Reliability in Cloud Deployment ... 20
5 Project Planning and Implementation ... 21
5.1 Defining System Requirements ... 22
5.2 Data Sources and APIs Used ... 22
6 Project Environment ... 23

6.1 Packages and frameworks 23
6.2 Installation and Configuration Details 24
6.3 Folder structure of the Application 26
7 Back-end Implementation 27
7.1 Entry Point of Back end module-index.js 27
7.2 Connecting to Azure Data Storage 31
7.3 Connecting to Google Analytics API 32
7.4 Connecting to Salesforce API 35
7.5 Database Setup with PostgreSQL 37
7.6 Data processing and Aggregation 38
7.7 Error Handling and Logging 40
7.8 Backend API Authentication with JWT 41
8 Front-end Implementation 43
8.1 Connecting the Frontend to the Backend 44
8.2 Handling User Interactions 46
8.3 Creating the Dashboard UI 48
8.4 Implementing Data Visualizations 50
9 Deployment 52
9.1 Containerization with Docker 52
9.2 Creating the Dockerfile 52
9.3 Building Docker Images 54
10 Deployment on Azure Container Apps 55
10.1 Setting Up Azure Container Registry 55
10.2 Configuring Azure Container Apps 56
10.3 Deploying the Application on Azure 58
11 Testing Implementation & Performance Measurement 59
11.1 Performance Measurement 61
11.1.1 Application-level 61
11.1.2 Production-level 62
12 Evaluation of the Application 64
13 Conclusion and Future Work 65

1 Introduction

The increasing reliance on data-driven decision-making in modern organizations, the development of efficient data reporting systems became necessity. As businesses grow, volume of data generated and utilized grows significantly, which makes manual reporting processes more time-consuming, inefficient, and prone to errors. Many organizations make use of Excel sheets to manually compile data from different platforms. This traditional method not only requires a lot of time and effort from personnel but also increases the risk of discrepancies and inconsistencies in data reporting. The purpose of this thesis is to build an automated data reporting application for the client organization of the commissioning company. Automated data collection is expected to enhance data accuracy, provide a real-time overview, and improve decision-making to support client company. The application will minimise manual effort for the commissioning company and ensure operational efficiency.

The primary focus of this thesis is the development and deployment of a data reporting application for a Property Management Company. By combining data from multiple sources, the application provides comprehensive overview of the company's operations. Specifically, Salesforce gives unit count data, Google Analytics provides page view counts for all units and condos, and the Azure Storage Account pulls records of how many customers are utilizing multi-factor authentication (MFA), ensuring accurate tracking and analysis. By consolidating these datasets, the system enables a comprehensive comparison of unit-specific visit trends, facilitating better decision making on data for business organisations.

The research investigates how the automated data collection and reporting application is being developed with PERN stack technologies and deployed in Azure cloud environment to contain its efficiency and scalability in data driven business operations.

To address this question, the project focuses on developing a web application with React and Node.js, utilizing APIs from Salesforce, Azure, and Google Analytics to combine multi-source data flawlessly. Furthermore, secure and scalable deployment will be guaranteed through containerization with Docker and deployment on Azure Container Apps and Azure Container Registry to ensure real-time data visualization for the end users.

By addressing these objectives, the project aims to provide valuable insights encompassing practical knowledge for designing, developing, and deploying applications using up-to-date tools, technologies, and industry best practices. The research will provide an in-depth evaluation of the challenges, benefits, and potential optimizations involved in automating data reporting processes and delivering scalable, secure, and efficient solutions for businesses.

1.1 Commissioning Organization

The thesis is commissioned by Mobimus Oy, operating under the brand name Osuria. The company specializes in giving digital secretary services for wide variety of business organizations. Primary focus of Osuria to secure file sharing for business-to-business operations. Currently Osuria has business with property management companies in Finland, but it intends to serve wide range of client organizations around the globe. Osuria’s platform aims to eliminate the use of email attachments by providing clients a centralized and safer location to send and share files. The platform is also complemented with features that improve customer interaction by making announcement to all or specific customer segments, initiate conversations, create tasks and schedule events.

1.2 Background and Motivation

Osuria handles data flow for customers on behalf of its clients, primarily property management companies (PMCs). There are multiple units in a condominium, and each unit is subject to ownership and occupation by different types of stakeholders, PMCs usually oversee condominiums. The data generation is huge and needs to be properly managed. Osuria unfolds details like how many units’ clients manage, page views at the PMC level as well as the condo level, and trends over the latter. Additionally, clients request information about the usage of Multifactor Authentication among their end-users.

Currently, Osuria delivers monthly reports to its clients, which involves the manual aggregation of data into Excel sheets. It entails gathering data independently from Salesforce, Azure, and Google Analytics, following a time-consuming approach. For instance, an employee must individually log into different platforms, extract the data, input the data into an Excel report, and send the report to the clients. This method works just fine for a small portfolio, but it is not scalable to a larger client base. The need for automation in data reporting is therefore evident.

The project aims to streamline the existing resource-intense manual reporting process with an automated data collection and reporting system. This will not only make it easier in terms of scalability and efficient processes but will also limit manual work. If successful, this automation solution could be mature enough that it could be sold to market, making Osuria's services better and generating more business. The long-term goal is to transform this automation solution into a marketable product, enhancing Osuria's service offerings and creating new business opportunities.

1.3 Problem Statement and Research Question

Integrating data manually across multiple services like Salesforce, Google Analytics, and Azure Storage proves to be an inefficient and unscalable process for organizations processing large volumes of records. As of now, the data is compiled in Excel sheets taking at least one hour for each report and does not make sense as the number of customers grows; This constant manual working approach is time-consuming, prone to human errors, and completely unsustainable which is where automation becomes much needed for operational efficiency and timely insights.

This research aims to investigate how automation can optimize, speed up, and scale in the cloud by automating a reporting system that brings data from different platforms and create real-time reports. Using modern development tools and architecture, the project intends to measure the viability and performance of such a solution.

The primary research question guiding this study is:
"How the automated data collection and reporting application is being developed with PERN stack technologies and deployed in Azure cloud environment to improve efficiency, accuracy, and scalability in data-driven business operations?"

To address this research question, the study explores the following sub-questions:

1. What are the key considerations and challenges in developing a data reporting application using the PERN stack?
2. How does API integration with Salesforce, Google Analytics, and Azure impact data retrieval and processing efficiency?
3. What are the benefits and challenges of deploying the application using Azure Container Apps and Azure Container Registry?
4. How does containerization with Docker improve the scalability and maintainability of the application?
5. How does real-time reporting can save cost, optimize resources, ensure data accuracy and consistency influence business decision-making and resource planning?

By addressing these questions, the study aims to provide valuable insights on creating and deploying cloud-based applications using cutting-edge tools and practices. The research is intended to contribute to the understanding of how automation further reduce errors, support data-based decision-making to improve business operations.

1.4 Research Methodology

The research emphasizing the development and deployment of an automated data reporting application by following a design and implementation-based methodology. The process starts along with a literature analysis to understand existing approaches to automated data collection, API integration, and cloud deployments.

To get the solution close enough to the business needs, requirements analysis is conducted by gathering input from Osuria employee who usually creates manual data report. This serves as reference points and helps define key functionalities and performance benchmarks while addressing technical limitations and security aspects of managing sensitive customer information.

The development phase involves building the application using the React, Node.js, Express.js, and Azure PostgreSQL flexible server. The application integrates APIs from various platforms such as Salesforce, Google Analytics, and Azure Storage Account.

The deployment includes designing the scalable architecture using Docker for Containerization followed by Deployment using Azure Cloud environment in particular utilizing Azure container Apps & Container registry.

1.5 Research Approach

This research applies a mixed method with which to comprehensively assess the automated data reporting system. The qualitative aspect focuses on integrating information through discussion and feedback sessions with Osuria personal, who have direct contact with their clients to explore challenges, and user expectations. Such qualitative insights assist to understand where improvements can be made and understand what the key facts are prompting users to adopt the solution.

On the quantitative side, the system performance is assessed using key metrics such as data processing speed, operations efficiency in comparison to manual work. Azure monitoring tools track metrics that can provide insight into the system's scalability and resource utilization. By combining qualitative and quantitative methods, the research offers a holistic understanding of the impact of automation on data reporting and business operations.

1.6 Scope and Limitations of the Thesis

The scope of this thesis is centred around the design, development, and deployment of a data reporting application using the PERN stack, Docker, and Azure cloud services. The focus of

the project is to automate the collection and reporting of customer data from Salesforce, Azure, and Google Analytics. This project covers backend development using Node.js and Express, frontend development using React, containerization with Docker, and deployment on Azure Container Apps. The scope involves integrating of APIs for data fetching and creating a unified dashboard for displaying the aggregated data. The application is intended to embed with Osuria's customer dashboard which is made by SharePoint technology. The app will appear as an icon or button which will not maintain any direct relation with SharePoint. The real-time Report will be visible to the customer upon clicking on the icon.

However, there are certain limitations to this thesis. The application is designed particularly for Salesforce, Azure, and Google Analytics, and may not be directly applicable with other data sources without any major modifications. The application requires limited user interaction. In addition, the security and performance optimization aspects are considered within the context of Azure Container Apps but may require further enhancements for larger-scale deployments. Advanced topics like microservices architecture or data mining with machine learning are not part of the thesis as the aim is to provide a solution for automated reporting.

1.7 Structure of the Thesis

This thesis is discussed considering three main parts: Theoretical Discussion, Project Implementation, and Deployment, each providing a structured, detailed, and comprehensive analysis.

The first part, Theoretical Discussion, explains and analyse an overview of the technologies used in the project, including the PERN stack, Docker, and Azure services. The necessity of automation in data reporting and the challenges with manual reporting processes are being discussed here in this phase of the thesis.

The second part, Project Implementation, covers the practical aspects of building the data reporting application. It analyses the backend and frontend development processes, the uses of APIs for data retrieval, and the design of the user interface.

The third part, Deployment, emphasizes on the containerization and deployment of the application. It explains how Docker is used to create image for the application and how it is deployed on Azure Container Apps using Azure Container Registry. The thesis concludes with an evaluation of the application, a summary of the findings, and suggestions for future change.

2 Theoretical Perspectives on Data Integration & Reporting

Modern organizations are badly dependent on effective data integration and reporting. Automated reporting ensures efficiency in data-driven decision making by reducing errors, and real-time insights. Integrating platforms like Salesforce, Azure, and Google Analytics enhances data collection and facilitate data analysis capability of the business organization. As manual reporting has issues regarding time-consumption and accuracy, web applications help mitigate through automation. A well-structured backend facilitates scalability and maintainability, ensuring seamless data integration and processing across multiple sources. This section explores theoretical perspectives in a deeper manner.

2.1 The Importance of Automated Data Reporting

In this context, automated data reporting not only improves the efficiency of organizations by limiting the amount of manual work but also minimizes the risk of errors in the data processing (CloverDX, 2023). Manual reporting processes can cause delays and inaccuracies that prevent timely decision-making (Alteryx, 2023). Automation provides real-time data collection, processing, and visualization from multiple sources, allowing business intelligence specialists to make data-driven decisions faster (Domo, 2023). In addition, automated reporting tools can be customized for specific business needs, creating more relevant analyses and insights (CloverDX, 2023). This is particularly important in industries where timely access to data can significantly impact performance and outcomes (PMC, 2013).

2.2 Overview of Salesforce, Azure, and Google Analytics as Data Sources

Salesforce, Azure, and Google Analytics are common platforms to get a crucial data for businesses. Salesforce as a cloud-oriented CRM platform that help businesses to maintain customer relationships, sales processes, and marketing campaigns. It provides data on customer behaviour, that is valuable for strategic decision-making (Salesforce, 2023a).

Microsoft Azure is a complete cloud solution built on a robust set of services, which include data storage, analytics, and machine learning. It offers scalable solutions to organizations managing and analysing large amount of data (Microsoft Azure, 2023).

Google Analytics, web analytics service, that tracks and reports website traffic and user behaviour. It provides insight into how users engage with their websites and localize their online strategies accordingly (Google Analytics, 2023). Integrating data from these platforms

into a single reporting system can greatly improve an organisation's ability to analyse and act on its data (Salesforce, 2023b).

2.3 Common Challenges in Manual Data Reporting Processes

Manual data reporting processes present a number of challenges that can have an adverse effect on organizational performance. The major problem is that, collecting data from many different sources and preparing them to report is very time-consuming. This often results in delayed critical insights, which can impact the decision-making process (CloverDX, 2023). The risk of errors and inconsistencies in the reporting of data can also be increased. Manual processes also involve human errors, risking the precision of reports (Alteryx, 2023). when businesses scale and have more data to process, manual reporting becomes less viable, as more resources and personnel are needed (Domo, 2023). Automation of these tasks could prevent such issues arising, saving businesses time, reducing errors, and ultimately improving reporting (PMC, 2013).

2.4 The Role of Web Applications in Solving Data Reporting Challenges

Web applications have proven to be the most appropriate solution for challenges that manual data reporting brings. By using modern web technologies, organizations can build dynamic dashboards that provide real-time access to key metrics and insights.

These applications enable easy integration of data from multiple sources, lessening the requirement for manual data entry and computation. Web applications are accessible from anywhere that provide flexibility and conveniences to the users. APIs enable web apps to integrate with other systems, such as Salesforce, Azure and Google Analytics, to guarantee that the data is always updated and correct. Web applications also include scalable capabilities, which enable organisations to process increasing amounts of data with minimal alterations to their infrastructure (Shklar, Rosen, 2009).

2.5 Data Source Integration - Modular Architecture

Modular architecture is a distinguishing feature of backend, which enables the system to aggregate with several data sources. It allows to maintain each module as independent but collectively contributing to the application through the modules. The backend demonstrates a high degree of maintainability, and scalability in its operation by isolating responsibilities into separate files. Different modules can be developed and tested independently, as they are independent of one another. This separation of concerns becomes especially beneficial in larger applications, where a team of developers might be working on various aspects of the

system simultaneously. Furthermore, this architecture facilitates scalability using different data sources and add additional modules without disrupting the existing workflow (Jog, 2016).

2.6 Security Considerations and Justification for JWT

In modern day web development, security has become the prime consideration while applications exchange sensitive data across networks. Keeping sensitive information in the client-side presents significant risks as frontend assets are publicly accessible that can expose secrets. Therefore, strong authentication and authorization are needed to protect backend resources and ensure secure communication between system components (OWASP, 2024).

Token-based authentication method is widely used for these purposes. JSON Web Tokens (JWT) is the major name among the accepted solutions. JWT is designed to transmit connection between parties. It offers several advantages over traditional static tokens or hardcoded API keys. Important thing is that JWTs are digitally signed from the server side. It uses a secret that securely stored and never exposed to the client. This security arrangements protect the token from attackers and ensure that only authorized clients are allowed to reach the backend services (Jones, 2015).

In this project, JWT is selected as the security mechanism due to its simplified suitability. Frontend uses JWT provided by the server while accessing the backend. It is short-lived but scalable in manner. That means it can be done in the runtime without a need of extra layer login in the frontend. This approach removes the need of using sensitive API keys or credentials in the frontend code that significantly eliminating the risk of unauthorized access.

JWT follows security best practices that allows limited life span and defining access scopes. If a token is compromised, the short life limits the potential chances of misuse. Backend also can control the routes or actions for the token that bring another layer of security. This design ensures secure data handling across networks (Stuttard & Pinto, 2011).

Overall, using JWT develops the security posture of the application without creating unnecessary complexity. It provides a balance between protection and ease of use, making it particularly appropriate for internal dashboards or trusted user environments. Moreover, using JWT, can keep the architecture flexible with future enhancements possibilities and it can add specific IP restrictions as well as access control while necessary.

2.7 Testing Methodology and Analysis of Results

Testing has a fundamental role software development. It ensures correctness, reliability, and security of application. As a systematic approach, testing identify defects, verify that whether the software meets the requirements or not. Testing validates the applications' performance according to the goal set in different conditions. Well-defined testing has great impact on the software quality by detecting early issues, reducing cost, and increasing user confidence in the system (Bierig, Brown, Galván, Timoney, 2022).

According to software functionality and behaviour, testing can be categorized into several types. Unit testing checks the individual components or functions to produce expected outcome. Integration testing examines the interactions between integrated components. It ensures that data flows correctly and combined units work together flawlessly. System testing checks the entire system and verify that whether the system meets pre-defined functional and non-functional requirements or not. Additionally, acceptance testing is related to the user. It confirms that the software satisfies user need before the final release (Aniche, 2022).

In case of web applications, security testing is considered as top priority testing. Security testing verify that unauthorized users cannot access restricted resources and vulnerabilities as injection attacks are properly mitigated. Security testing identifies the weaknesses of the system that creates room for the developer or testers to deal the loopholes.

Automated testing frameworks are the key to modern testing practices. Jest, React Testing Library, and Supertest support the automation of unit and integration testing in JavaScript and Node.js environments. Mocking techniques is a notable method that isolate components from external dependencies during testing. For completeness, the level of test coverage in software testing determines the combined code or functionality exercised during testing. A good test suite has both positive tests, which ensure the system works correctly, and negative tests, which ensure the system fails appropriately (Mili and Tchier, 2015).

In summary, software testing is an important phase in the development lifecycle, guided by established methodologies and supported by various tools and frameworks. In this project, these principles form the basis for the testing strategy applied to both frontend and backend components, ensuring that the application meets industry standards for quality, reliability, and security.

3 Technologies Used in the Project

The project utilizes popular modern technologies to create an efficient and scalable data reporting application. PostgreSQL, Express. js, React. js, and Node. js together constitutes the basis of back and front-end development. Azure cloud services used for cloud deployment while Docker container ensures the portability of the application in different environments. REST APIs connect to external data sources like Salesforce, Azure, and Google Analytics. Authentication enables data security to ensure reliable reporting system.

3.1 Overview of the PERN Stack

The PERN stack, consist of PostgreSQL, Express.js, React.js, and Node.js, is used for full-stack web development. React.js serves as the front-end library, which helps developers to create dynamic and interactive user interfaces. The back end relies on Node.js, a JavaScript runtime, along with Express.js, a lightweight web application framework, to create RESTful APIs as well as handle server-side logic. PostgreSQL, an open-source relational database, is utilized for data storage, offering a structured and scalable solution for managing data-intensive applications. Front-end and back-end are tightly coupled within this stack, leading to the development of fast, efficient and highly responsive web applications. (Alves, 2023).

Due to some complexity, PERN stack is comparatively less flexible than MERN. PERN is not ideal in terms of strong relational data handling, but it is unparallel for structured data growth.

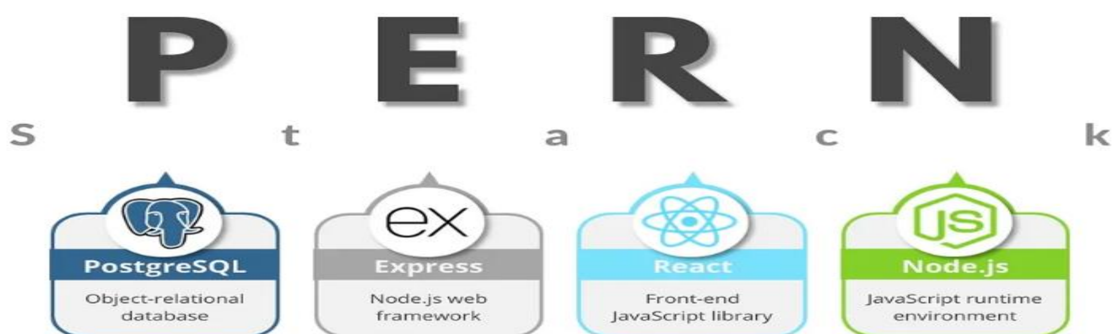

Figure 1. PERN stack architecture (Alves, 2023)

The PERN stack simplifies development with JavaScript throughout the application, both on the client and server-side. React.js enables component-based UI design, which facilitates code reusability and maintainability. Express.js and Node.js are used to shorten middleware integration and asynchronous operations, so that the server-side can be highly scalable. PostgreSQL uses pooling libraries like pg-pool for complex queries and transactions,

combines efficiently with the Node.js environment, allowing developers to manage data effectively while maintaining integrity of data. These features make the PERN stack an excellent choice for modern, scalable applications (Pawar, 2025).

3.2 PostgreSQL: Database for Storing Data

Azure PostgreSQL Flexible Server is a Microsoft Azure managed database service that allows to manage PostgreSQL, which is an open-source relational database system. Azure PostgreSQL started its journey in November 2021 to offer developers flexibility in deployment, and control over their workloads and performance. PostgreSQL is a relational database technology that can efficiently manage, store, and retrieve structured data, making it suitable for a variety of use cases. Flexible Server is highly optimized for application with high availability, scalability, and cost optimization. Its ability to handle complex queries and diverse data types enhances its versatility for modern application development. (Microsoft, n.d.)

Azure PostgreSQL Flexible Server interacts with relational data using SQL with a structured and efficient way. It maintains data integrity and system reliability by automated backups, high availability zones, and monitoring tools. A major advantage is scalability; developers can scale resources independently as data grows in volume, optimizing costs in the process. In addition, the platform prioritizes security of data at the rest and in transit and only allowing authorized users to access or modify the database. Integrating with Azure Active Directory reduces the complexities surrounding identity and access management. Overall, Azure PostgreSQL Flexible Server is a secure and scalable solution for variety of application requirements (Vadlamani, 2024).

PostgreSQL has strict schema that can slowdown rapid development. It is not easier to configure and maintain regarding availability and performance. PostgeSQL toolkit is not included with the base distribution of its architecture that is why these extensions may need additional expertise to set-up and maintain (Kolovson, 2021).

PosgreSQL is efficiently managed in this project. PostgreSQL stores siteid that is used in the frontend query parameters. When backend API receives request to /data endpoint, it queries over PostgreSQL database to fetch stringfiltervalue, which is then used to request analytics data from Google Analytics.

3.3 Express.js: Backend Framework for API Development

Express.js is a minimal and flexible web application framework for Node.js. It is designed to develop of single-page, multi-page, and hybrid web applications. Express simplifies server-side development through middleware that manages the request-response cycle and robust

routing capabilities. With support for RESTful API design, Express handles GET, POST, PUT, and DELETE methods as different HTTP requests. Since the first release in 2010, Express has become most widely used in node.js ecosystem (Satheesh, D'mello, and Krol, 2017).

Despite being popular, Express still lacks built-in security features. Developers need to manually implement protections against SQL injection and cross-site scripting (Purewal, 2014). Fastify is claiming better performances over Express recently.

In this project, Express.js is used to create the backend API, handle incoming requests, and serve data to the frontend. It manages routing, processes query parameters such as siteId, and integrates middleware for request validation. The framework ensures seamless communication between the frontend and backend by handling API requests, fetching data from external sources like databases and third-party services, and returning structured responses to the client.

3.4 React: Frontend Framework for Building User Interfaces

React, as a JavaScript library developed by Facebook in 2011 for developing dynamic and interactive user interfaces. React has component-based architecture that encourages the development of modular and reusable UI components, allowing maintainability and scalability in large-scale applications. The virtual DOM is one of the major innovations of React, and it boosts performance by reducing direct manipulation and engagement with the real DOM, providing fast and efficient updates (Sengupta, Singhal, and Corvalan, 2016).

In this project, React is used to build a dashboard that displays aggregated data from Salesforce, Azure, and Google Analytics as sources in a visually appealing format.

React has two types of components, such as functional components, and class components. Functional components are stateless only and usually used for returning UI based on props. Class components, on the other hand, are stateful and can use lifecycle methods like componentDidMount and componentWillUnmount.

Despite its advantages, React can challenge beginner to deal with advance features such as hooks, state management libraries like Redux, and context APIs. In absence of proper handling with state or props may lead performance bottlenecks while managing larger applications. React.js uses too many third-party libraries that negatively impact on load times. Too many small reusable components can create over fragmentation. Prop drilling passes data through multiple layers that create extra layer of complexity (Sharma, 2024).

However, with React Hooks, functional components can handle state and side effects, leading to their increased popularity in modern development. React is particularly popular for it's featured to enable single-page applications. It helps developers to optimize the management of application state and rendering of components to improve user experiences (Hinkula, 2023).

3.5 Node.js: Runtime Environment for Backend Services

Node.js is an open-source, cross-platform set of runtime environments based on Google Chrome's V8 JavaScript engine. It enables developers to run JavaScript code on the server side, making it possible to develop scalable and high-performance applications. Unlike traditional multi-threaded request-response models, Node.js utilizes a single-threaded event loop architecture, which make sure non-blocking I/O operations. This characteristic adapts it well to real-time, push-based architectures like chat applications or collaborative tools. Node. js is lightweight runtime environment, not a framework like the others mentioned above, which is focused building applications fast (Herron, 2016).

However, Node.js may not be well-suited for CPU-intensive tasks due to its single-threaded nature, which can become a bottleneck in certain workloads. As like React, Node.js uses multiple third-party packages that increases the risk of security vulnerabilities (Brains, 2024).

As default, Node.js runs backend server in this project using Express.js to handle API requests from the frontend. It connects to the external services like, PostgreSQL, Google Analytics, Azure Table Storage, and Salesforce to have seamless dataflow. It collects data, process the collected data, and return the processed data to the frontend UI.

Modules in Node.js play a crucial role in application development by compressing functionalities within reusable code blocks. These modules can be categorized into core modules, local modules, and third-party modules. Core modules, like 'fs' for file system operations or 'http' for server-side functionality, are built into the Node.js runtime. Local modules are created by developers to handle specific application logic, while third-party modules are distributed via npm, the Node.js package manager. This modular architecture encourages code reuse and easier control of the app's dependencies (Powers, 2012).

3.6 Docker: Containerization Technology

Docker is an open-source platform used by developers to automate the deployment of applications using lightweight, portable containers. Containers -- package an application and its dependencies, and guarantee that it works as expected in all environments.

This resolves the traditional problems of environment differences between development, test and production (Martin, 2024).

Docker provides advantages such as portability, scalability, and consistency in application development. Containers enable developers to create applications that contain everything they need to run, minimizing compatibility headaches. The light weight of the Docker also leads to faster deployment and better use of resources (Loubser, 2021).

However, the adoption of Docker brings some complexities regarding security. For instance, when virtual machines become heavier than containers, they still share host system's kernel that need to be negotiated properly. Additionally, while scaling, container coordination with Kubernetes may add operational overhead for devOps practitioners and small-scale team (Quadri, 2024).

In this project, Docker will make sure that the data reporting app is consistent in deploying environment. Building a Docker image makes it much easier to produce the application for use on Azure Container Registry and Container Apps, allowing for simplified deployment and better maintainability that ensures better performances from the application.

A Dockerfile is a script with a series of commands that are used to create a Docker image. It describes the base image, application dependencies and commands copied inside the container. For academic use, the Dockerfile is effectively the specification for reproducible builds. With a Dockerfile defining the instructions to create an image, it allows developers to ensure the same image is created no matter where it is built. This removes the traditional "it works on my machine" issue in software development.

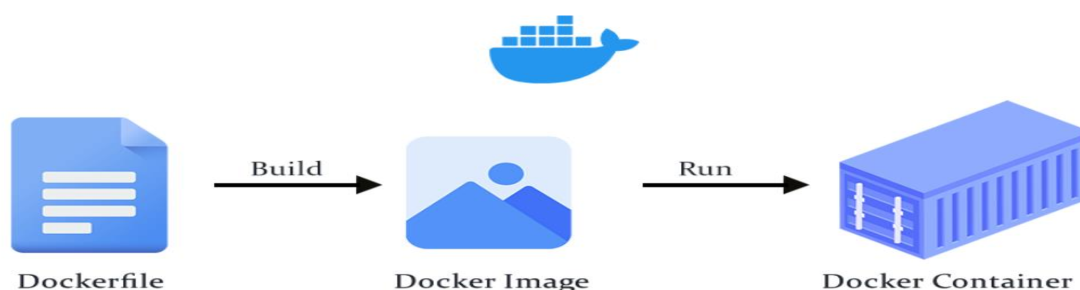

Figure 2. Docker build and run process (Ayanda, 2023)

Containerization, Build the Docker image from the Dockerfile, push it onto Azure Container Registry, and deploy it on Azure Container Apps. A container image can be deployed in various environments, which means that this process creates a reliable and constant version of the application throughout the software development lifecycle (Agarwal, 2021).

3.7 Azure Cloud Services

Azure is a cloud computing platform and service created by Microsoft that can be integrated into applications. Azure offers scalable and secure infrastructure and reliable foundation to run applications in the cloud (Savill, 2020).

Azure Container Apps is a container service that lets developers to deploy their containerized applications without maintaining the infrastructure. With auto-scaling by HTTP requests, CPU, or memory usage, Container Apps ensures that applications remain responsive under variable workloads. In this project, the data reporting application is being deployed using Azure Container Apps to serve as a cloud deployment solution for the Dockerized application (Haakman and Hooper, 2023).

Container App doesn’t have direct access to Kubernetes APIs. It has little control over infrastructure and no automatic scaling for web applications.

In this project, the Docker image created utilizing the Dockerfile is pushed to Azure Container Registry. It is a Microsoft provided private Docker registry service. It allows developers to store and manage container images securely and ensure that it is readily available for deployment on Azure Container Apps. The integration between Azure services makes it relatively easier to deploy and secure the services (Ifrah, 2023).

3.8 REST APIs: Connecting to Salesforce, Azure, and Google Analytics

REST API (Representational State Transfer Application Programming Interfaces) connect the application with external data. This project connects REST APIs to fetch information from Salesforce, Azure and Google Analytics. These APIs allow data to be integrated into the reporting application in such a way that the dashboard is constantly reflecting real-time data that is up to date and accurate.

Data security is a critical aspect of application development, particularly when dealing with sensitive customer information. In this project, several security measures are implemented to protect data from unauthorized access. Siteid and partition key needs to be provided as a query string to access the application. With this set-up the data will be only accessible to the right users and disallow unauthorized access (Brown, 2014).

4 Application Deployment Concepts

Application deployment involves hosting software in an environment what is dynamically accessible, scalable, and reliable. Application deployment can be divided into two main categories such as On-premises deployment and cloud-based deployment. Traditional deployment approach provides high control and better security, but it is not cost-efficient for businesses. On the other hand, cloud ensures scalable, flexible, and cost-effective deployment option that need minimal maintenance from the end user perspective. This project is deployed in Azure cloud environment for better reach capability.

4.1 Understanding Cloud Deployment

Cloud deployment of the application refers to the process of hosting and managing an application on the cloud infrastructure that is accessible over the internet. This includes taking a cloud environment, configuring the required resources, and deploying the application to achieve scalability, reliability, and security. Cloud deployed applications take advantage of the scalability, flexibility, and availability of cloud services so that they are available from anywhere and scale to demand. Cloud applications typically utilize a combination of storage, computing power, and networking capabilities from cloud services like Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform. The main advantage of this method is that it can allocate resources better and decrease the infrastructure expenses enabling the applications to deliver better performance (Wittig, Wittig, 2015).

The developed data reporting application in the context of this thesis is deployed on Azure, utilizing cloud services including Azure Container Apps and Azure Container Registry. These services allow the application to manage increasing data volumes and guarantee users with persistent access to real time reporting dashboards, aside from hardware restrictions and maintenance tasks (Chao, 2013).

4.2 The Role of Containerization in Modern Software Development

Containerization is a widely used practice nowadays that empowers developers to pack applications and their dependencies into portable, lightweight containers that can run uniformly on any environment. Containers ensure that the application works the same way in development, testing, and production environments, reducing compatibility issues caused by differences in system configurations (Agarwal, 2021).

Docker, the most popular containerization platform, has changed the way applications are built, ship, and run. Docker prints standardized environment for the application that developers can Build so that the deployment becomes easier and the chance of error becomes less. Containers also allow isolation between them making sure that the applications do not interfere with each other, even when running multiple services on the same host (Loubser, 2021).

Docker is used in the project to containerize the data reporting application, allowing it to be moved around easily and deployed smoothly on Azure Container Apps. Docker images are built and pushed to Azure Container Registry. Container Apps uses those images from the container registry. Docker works in the back end, assuring the application remains same in terms of various environment types, thus facilitating scaling and maintenance (Martin, 2024).

4.3 Scalability and Reliability in Cloud Deployment

As cloud is a distributed architecture, where application/service has run on multiple servers, so Scalability and reliability are crucial for its success in cloud deployment. Scalability is the ability for an application to size resources according to demand, using more resources during peak usage and less resources during low usage to minimize costs (Microsoft, 2022). While reliability guarantees that the application stays accessible and functional, even during hardware malfunctions or unforeseen traffic surges (Katzer, 2020).

With Azure Container apps, built-in scalability provides this feature so the data reporting application can automatically scale depending on the number of requests it is receiving. This guarantees scalability of the application without compromising performance as the workload grows. Moreover, the reliability features of Azure (load balancing and failover mechanisms, for example) provide high availability so that users are still able to access the application despite server failure (Haakman and Hooper, 2023).

By leveraging cloud deployment services, organizations can achieve high availability and performance for their applications while reducing the complexity of managing infrastructure. In this project, we can see that Azure Container Apps enables the data reporting application to be remained scalable, reliable, and efficient, which fulfils the requirements of users and businesses (Microsoft, 2022).

5 Project Planning and Implementation

Success of a software development project badly depends on the effective planning of the project. In this project, the planning phase includes identifying the key tasks, milestones, and timelines needed to develop the data reporting application utilizing the PERN stack and deploy it on Azure. The project timeline is divided into several phases, including requirement gathering, backend and frontend development, containerization with Docker, and deployment on Azure Container Apps. The timeline is structured as follows:

Week 1–2: Requirement gathering and system design
Defined application goals, determined data sources (Google Analytics, Salesforce, Azure), and designed the system architecture for modular data reporting.

Week 3–6: Backend development using Node.js and Express
Built a REST API in index.js with secure data retrieval routes and integrated external services like Salesforce, Google Analytics, and Azure Table Storage.

Week 7–9: Frontend development using React
Developed components such as TableData.js, Comparison.js, and ExportToPDF.js, and implemented query parameter handling using useQuery() in App.js.

Week 10–11: Database integration with PostgreSQL
Created and configured the db.js module to fetch site details from Azure PostgreSQL and linked it to the analytics route.

Week 12–13: Dockerizing the application
Wrote Dockerfiles for both frontend and backend, and tested builds locally using Docker Compose to ensure container compatibility.

Week 14: Testing and validation
Jest Unit testing and React testing library are being used. Supertest and Jest are used in the backend to test Express.js API endpoints.

Week 14: Deployment on Azure Container Apps
Deployed frontend and backend containers using Azure Container Apps and Azure Container Registry and monitored performance via Azure Metrics.

This unique method ensured a structured completion of each phase within the specified timeframe, that minimized the risks of delays and ensured a seamless development process.

5.1 Defining System Requirements

The requirements for this project are defined based on the functionality needed to automate reporting of data from Salesforce, Azure and Google Analytics. The requirements are divided into two main categories such as functional requirements and non-functional requirements.

Functional Requirements:

1. The system has to retrieve data from Salesforce, Azure, and Google Analytics using their respective APIs.
2. The system needs to use PostgreSQL database to store query parameter data.
3. The frontend must display the data in a user-friendly dashboard using React.
4. **The system must allow clients of Osuria to filter, search, and create data reports.**
5. The system must provide secure access to authorized users.

Non-Functional Requirements:

1. The system must be scalable to handle an increasing volume of data.
2. The system must ensure data security and user authentication.
3. The system must be portable and deployable using Docker containers.
4. The system must be deployed on a reliable cloud platform (Azure).

5.2 Data Sources and APIs Used

Salesforce, Azure, and Google analytics are the three main data sources utilized in the project. Each data source has its own API, which is integrated into the backend of the application to automate data retrieval.

Salesforce API: The REST API provided by Salesforce enables developers to retrieve customer data, such as account information, leads, and sales performance. Customer data is retrieved via API for reporting (Salesforce, 2022).

Azure API: Microsoft Azure provides a range of APIs to work with its services, including Azure Storage, Azure SQL Database and Azure Table Storage. The Azure API is utilized in this project for storage account uses and data retrieval for reporting purpose (Microsoft, 2022). Google Analytics API: Google Analytics provides access to data around website traffic, user behaviour metrics, and performance reports. This API is then used to fetch the required analytics data for inclusion in the reporting dashboard (Google Analytics Help Center, 2022).

6 Project Environment

Setting up the environment is the first and foremost step to create a web app project. An effective, productive, and time-saving project heavily relies on the effective utilization of the tools and stack together. The whole project is done in windows operating system. There are a number of software and third-party tools being used in developing the project. The project utilized Visual Studio Code as the coding editor of choice as an open source project led by Microsoft that was first released in 2015. It has since become one of the most popular code editors for developers. VS Code is a great toolkit which has setup, UI for displaying content of the files, settings to be customise, debug a NodeJS web program and cloud deployment features of the web app.

6.1 Packages and frameworks

The frontend of the application is built with React.js (v18.3.1) to create user interfaces. React-DOM (v18.3.1) is used with React to render components into the DOM effectively. To minimise unnecessary re-rendering and for seamless navigation in the client-side routing, React Router DOM (v6.24.1) has been incorporated with the application.

For application functionality testing, React Testing Library (v13. 4. 0) and for unit testing, Jest (v 5.17.0) is being used in the application. For user interaction simulation, the application is using complementary packages such as @testing-library/jest-dom and @testing-library/user-event. The Azure MSAL Browser library (@azure/msal-browser, v3.18.0) is integrated for authentication and authorization, which accelerates secure user login processes for the application via Azure Active Directory. Axios version (v1.7.0) is used for making HTTP requests to the backend API and making asynchronous data fetching easier.

React-scripts (v5.0.1) gives necessary configurations for development and build operations. Performance metrics are ensured using web-vitals (v2.1.4). Additionally, the frontend's development server has been set with a proxy to forward API calls to http://localhost:3002, to ensure proper backend connection during local development.

while Azure services were seamlessly connected using @azure/identity for secure authentication. Additionally, Google Analytics was integrated using @google-analytics/data and @google-auth-library.

6.2 Installation and Configuration Details

The initial phase of the project begins with the initialization of Node.js to create a basic framework for managing dependencies. Running npm init -y generates a package.json file that stores all relevant metadata related to the project. Express.js is included for backend routing while dotenv securely stores environment variables in the application. Cors is being used for safer communication between client and server from different origins. In the development phase, nodemon package is used to automatically restarts the server after any changes happened in the codebase. Pg package integrates PostgreSQL database using JavaScript. With this package, SQL queries are being executed within the Node.js application. It enables the application to store and retrieve data efficiently from a PostgreSQL database. Azure Identity package provides secure authentication and data handling. Google Analytics is incorporated with couple of packages, such as @google-analytics/data and @google-auth-library while Salesforce is integrated with this application with Jsforce library. All required dependencies can be installed with the following command:

```
npm install express dotenv cors nodemon pg @azure/identity @google-analytics/data @google-auth-library jsforce
```

This streamlined installation process ensures all essential packages are included efficiently.

On the front-end side, npx create-react-app is used to initialize the application. The core structure of the app is built using React, while react-dom handling the user interface. For consistent routing between different sections, React Router DOM is included. API requests are managed with Axios, while react-scripts provides essential scripts for development and builds. The testing environment set up is done with React Testing Library, along with Jest DOM for asserting the Document Object Model (DOM) and User Event for simulating and testing user interactions. All required dependencies are installed with the following command:

```
npm install react react-dom react-router-dom axios react-scripts @testing-library/react @testing-library/jest-dom
```

Additionally, authentication is handled through the Azure MSAL Browser package, ensuring a smooth login experience and seamless integration with Azure Active Directory.

Configuration is done with connecting the front-end and back-end using a proxy, setting up environment variables in a.env file for the keeping of sensitive information such as database credentials. With all installations and configurations done, backend runs with nodemon index.js and frontend development starts with npm start to get the ball rolling.

```
{
  "name": "unified-dynamic-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@azure/data-tables": "^13.2.2",
    "@azure/identity": "^4.3.0",
    "@google-analytics/data": "^4.7.0",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5",
    "express": "^4.19.2",
    "google-auth-library": "^9.11.0",
    "jsforce": "^1.11.1",
    "jsonwebtoken": "^9.0.2",
    "nodemon": "^3.1.4",
    "pg": "^8.12.0"
  }
}
```

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@azure/identity": "^4.3.0",
    "@azure/msal-browser": "^3.18.0",
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^1.7.2",
    "html2canvas": "^1.4.1",
    "jspdf": "^2.5.1",
    "react": "^18.3.1",
    "react-datepicker": "^7.3.0",
    "react-dom": "^18.3.1",
    "react-router-dom": "^6.24.1",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  ▷Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "proxy": "http://localhost:3002",
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
```

Figure 3. Project dependencies, package.json

In this application development, Azure Database for PostgreSQL Flexible Server is used as the database and the local pgAdmin is used for database operations. Application has used existing Azure account that includes PostgreSQL flexible server on the Azure portal. Database user with the read and write access is established that to perform operations. Firewall rules were also updated to allow the application's IP address access to the database server for secure communication.

6.3 Folder structure of the Application

This is followed by setting up the initial environment by installing relevant tools and packages that will serve as the foundation for subsequent coding and implementation of the application. Folder structure is critical to establishing a clean and modular file structure that would enhance development to come and future extensibility. The layout of the application's files and directories to the appropriate folders are shown in Figure below, along with a clear separation between the frontend and backend.

```
client/
├── node_modules/
├── public/
├── src/
|     ├── components/
|     |     ├── Comparison.js
|     |     ├── ExportToPDF.js
|     |     └── TableData.js
|     ├── services/
|     ├── App.js
|     ├── App.test.js
|     ├── index.js
|     ├── App.css
|     └── index.css
├── .env
├── Dockerfile
├── package-lock.json
├── package.json
└── README.md

server/
├── node_modules/
├── .env
├── auth.js
├── azure.js
├── db.js
├── Dockerfile
├── ga4.js
├── index.js
├── jwt.js
├── keys.json
├── package-lock.json
├── package.json
└── salesforce.js
```

Figure 4. Folder Structure of the project, Source: OpenAI (2025).

The application architecture is divided into two directories, client and server. The client directory contains all the files and folders needed to generate the frontend, which is built using ReactJS. On the other hand, the server directory is designed to manage the backend functionalities, including database interactions, API integrations, and service configurations. It provides a clear separation for the roles of the application making them easily maintainable.

The important aspect of the application's design is the inclusion of .env files in the root of both the frontend and backend directories. These files safely store sensitive information, such as API keys and database credentials, safeguarding the integrity of the application during development and deployment. Additionally, package.json file records of dependencies and scripts, allowing an environment of the application to be consistently regenerated. The architecture of this application is carefully designed with a mix of simplicity and scales. This separation of concerns facilitates a smoother development process and offers a strong structure for potential growth, updates, and maintenance later.

7 Back-end Implementation

The back-end of the application collects data from different external data sources such as Azure Table Storage, PostgreSQL, Google Analytics 4, and Salesforce. In this application, modular approach of programming is maintained to keep all the data-fetching logic instead of single monolithic script. In this principle, each module captures specific logic and exposes only the necessary interfaces for interaction. This design choice minimizes complexity, avoids tight coupling, and simplifies debugging.

```
server/
├── node_modules/
├── .env
├── auth.js
├── azure.js
├── db.js
├── Dockerfile
├── ga4.js
├── index.js
├── jwt.js
├── keys.json
├── package.json
└── salesforce.js
```

Figure 5. Backend folder structure, Source: OpenAI (2025).

The picture shows the folder structure for the back-end implementation. It organized itself within the server directory. After node_modules there is a .env file that is used to store environment variables derived from Azure, Salesforce, and Google analytics. The azure.js, ga4.js, and salesforce.js are individual module to interact with its own data source. The db.js file is used for connecting to the PostgreSQL database. The keys.json file includes all the keys related to Google analytics service account and finally the index.js file is a convention in node.js for defining the main entry point for the back-end module.

7.1 Entry Point of Back end module-index.js

Once client makes a request to the backend, the index. js entry point that calls three different top-level modules. So, each module interacts with the respective data source to read one or more pieces of information or process some necessary information and then send the response back to the central controller in index. js. This modular approach ensures that the backend can handle the following tasks simultaneously:

- Retrieve site-specific configurations from PostgreSQL.

- Fetch analytical metrics from Google Analytics.
- Query authentication data from Azure Table Storage.
- Get account-related information from Salesforce.

The asynchronous capabilities of Node.js allow these modules to run parallely, maximizing efficiency and reducing response time.

```
const dotenv = require('dotenv');
dotenv.config();
const express = require('express');
const cors = require('cors');
const jwt = require('jsonwebtoken');
const ga4 = require('./ga4');
const azure = require('./azure');
const salesforce = require('./salesforce');
const { fetchSiteDetails } = require('./db');
const authenticateToken = require('./auth');

const app = express();
const PORT = process.env.PORT || 3002;

const corsOptions = {
    origin: 'http://localhost:3000',
    credentials: true,
};

app.use(cors(corsOptions));

app.use((req, res, next) => {
    const start = Date.now();

    res.on('finish', () => {
        const duration = Date.now() - start;
        console.log(`[METRICS] ${req.method} ${req.originalUrl} - Status: ${res.statusCode} - Time: ${duration}ms`);
    });

    next();
});

app.get('/get-token', (req, res) => {
    const JWT_SECRET = process.env.JWT_SECRET;

    const token = jwt.sign({ app: 'dashboard' }, JWT_SECRET, { expiresIn: '1d' });
    res.json({ token });
});

app.get('/data', authenticateToken, async (req, res) => {
    try {
        const siteId = req.query.siteId;
        const partitionKey = req.query.PartitionKey || siteId;
        const name = req.query.name;
        const month = req.query.month;
        const year = req.query.year;

        if (!siteId || !name) {
            return res.status(400).send('siteId and name are required');
        }

        const siteDetails = await fetchSiteDetails(siteId) || {};
        const stringFilterValue = siteDetails.stringfiltervalue;

        const visits = await ga4.getVisits(stringFilterValue, month, year) || 0;

        const previousMonth = month === 1 ? 12 : month - 1;
        const previousYear = month === 1 ? year - 1 : year;
        const previousMonthVisits = await ga4.getVisits(stringFilterValue, previousMonth, previousYear) || 0;

        const strongAuth = await azure.getStrongAuth(partitionKey) || [];

        let strongAuthIncludedToContract = 0;
        let strongAuthMade = 0;
        if (strongAuth.length > 0) {
            strongAuthIncludedToContract = strongAuth[0].StrongAuthIncludedToContract;
            strongAuthMade = strongAuth[0].StrongAuthMade;
        }
```

```
        const units = await salesforce.getUnitsByName(name);
        const visitsPerUnit = units ? (visits / units).toFixed(2) : null;
        const visitChange = previousMonthVisits ? ((visits - previousMonthVisits) / previousMonthVisits * 100).toFixed(2) : null;

        res.json({
            units,
            visits,
            visitsPerUnit,
            visitChange,
            strongAuthIncludedToContract,
            strongAuthMade,
            previousMonthVisits
        });
    } catch (error) {
        console.error('Internal Server Error:', error.message);
        res.status(500).send('Internal Server Error');
    }
});

// Error handling → capture all errors
app.use((err, req, res, next) => {
    console.error(`[ERROR] ${req.method} ${req.originalUrl} - ${err.message}`);
    res.status(500).send('Internal Server Error');
});


app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Figure 6. Backend server file (index.js)

The index. Js, which describes how to create a dynamic server with node. js and Express to respond to a request for data from a third party. Initially, using the dotenv package, environment variables are configured. The dotenv. config () method reads the. env file and makes the environment variables defined in that file available to the process, so they can be accessed from anywhere in the application. Please note this is a very important step as it helps the application to securely store/stay connected to secure information like API keys, DB connection strings, port numbers etc.

Express is used to set up the server to handle HTTP requests and responses as an API. CORS (Cross-Origin Resource Sharing) is for cross-origin requests and custom modules for interacting with Salesforce, GA4 (Google Analytics 4), and Azure services. The CORS Policy also supports the credentials which is sending cookies along with HTTP Authentication in cross-origin requests. The database module also imported so that the site specific details could be fetched that would be very important to build the response data. These dependencies are essential to function the application, which enable to get data from multiple external sources.

As it is shown in the code that once all the required dependencies are imported, the express application is initialized, and the port number using environment variable with a default fallback of 3002. The design of the application allows to deploy dynamically across multiple deployment environments. After initializing the app, CORS is added to ensure that backend is only accepting requests from certain frontend URLs, preventing untrusted clients request to

communicate with the backend. If the domain of frontend application changes, then this configuration also needs to change accordingly to avoid access issues.

The application sets up a GET route at the /data path, making it the main entrypoint where clients come to request data. The server parses the request to obtain parameters like siteId, name, month, and year. If there is no partitionKey is provided, it is also derived from the siteId. A validation check is performed to ensure that the required parameters siteId and name are present; otherwise, it responds with a 400 Bad Request status code, indicating to the client that the values are missing.

After validation of this process is successfully completed, the server creates multiple variables to store data extracted from the different services. These statistics include default values as site details, business units, website visits, and authentication-related parameters. Fetching site details, its done through a function named fetchSiteDetails, it queries the database using provided siteId. The details retrieved from the API, if any, will be available in the siteDetails variable; otherwise, an empty object is used to siteDetails to avoid errors in later processing.

The application is then executing a call to Salesforce to get how many units are there against the name. This is important information to know for calculating performance metrics like visits per unit. The server then makes a request to Google Analytics with the filter value of the site and the month and the year to get the number of visits after getting the units. The visit data from both current and previous months are retrieved but if there is no data is found, data defaults to zero.

Another step is related to partition key used to query Azure storage account for info related to authentication. These include values showing the number of contracts related to multifactor authentication, and how many users have implemented the strong authentications for security.

After collecting all the required data, the app performs calculations to provide meaningful insights The visits per unit metric is derived by dividing the total units, rounding to two decimal places. The result is set to null in case the number of units is zero, to avoid division by zero. In the same way, visit change percentage is calculated to show the growth or decline of visits over the last month. This value is set to null if there is no data for previous month.

With all data prepared and calculations performed, the response object is created, containing all information such as units, visits, visit change percentages, and authentication data. The response is then sent back to the client in JSON format, allowing the frontend application to easily consume this data for visualization or further analysis.

Finally, the server starts listening on the defined port and logs a message confirming that it is running and ready to handle incoming requests. This indicates that the setup is complete with the server running and ready to accept incoming data queries. Once a request is received, server checks for the presence of essential parameters like siteId and name. If either is missing within this system, the server instantly responds with a 400 Bad Request.

7.2 Connecting to Azure Data Storage

The azure.js module is for connecting with Azure Storage Account Table. It provides secure, and passwordless authentication. This module handles partition-based queries. The code creates a secure connection with Azure Table Storage using the @azure/data-tables library, leveraging @azure/identity for Managed Identity authentication. This dynamically obtain access tokens through Azure Active Directory and eliminates the need for hardcoded credentials. The script imports required dependencies, like dotenv for managing environment variables, maintaining flexibility in configuration. It creates a ManagedIdentityCredential instance, to authenticate seamlessly within Azure-hosted environments like storage account. In this approach, the application can fetch, add, and modify data in Azure Table Storage with ensured cloud security and data handling.

```
const { ManagedIdentityCredential } = require('@azure/identity');
const { TableClient } = require('@azure/data-tables');
require('dotenv').config();

const credential = new ManagedIdentityCredential();

const tableName = process.env.AZURE_TABLE_NAME;
const accountName = process.env.AZURE_STORAGE_ACCOUNT_NAME;

const tableClient = new TableClient(
    `https://${accountName}.table.core.windows.net`,
    tableName,
    credential
);

exports.getStrongAuth = async (partitionKey) => {
    try {
        const filter = `PartitionKey eq '${partitionKey}'`;

        const entitiesIter = tableClient.listEntities({ queryOptions: { filter } });
        const entities = [];

        // Log each fetched entity
        for await (const entity of entitiesIter) {
            entities.push(entity);
        }

        if (entities.length === 0) {
            console.log(`No entities found for PartitionKey: ${partitionKey}`);
        }

        return entities;
    } catch (error) {
        console.error("Failed to fetch data from Azure Table Storage", error.message, error.stack);
        return [];
    }
};
```

Figure 7. Connecting with Azure Table Storage

The picture represents a TableClient to interact with Azure Table Storage. The endpoint URL for the storage account is dynamically constructed using the AZURE_STORAGE_ACCOUNT_NAME environment variable, and the table to be queried is defined by the AZURE_TABLE_NAME variable. This configuration ensures that the application is dynamically adaptable to storage accounts without requiring changes to the source code. The TableClient is initialized with the ManagedIdentityCredential, to ensure authentication.

The main functionality here is within the getStrongAuth function which is an async function that get the entities from the provided azure Table Storage table according to the provided partitionKey. This function acts as a bridge layer between the application and the underlying data source. When called, it builds a filter query targeting to entities whose PartitionKey matches the provided value. This filter query is then used to query the table through the tableClient.listEntities method.

Inside the getStrongAuth function, the script loops through the entities returned by listEntities using a for await loop. Each entity then pushes into an array, that will eventually contain all the matching records with the given partitionKey. If no entities are matched, a log message is generated that no results are found for the query.

The function includes a robust error handling through a try-catch block, logging errors and ensuring the function returns an empty array in case of failure, preventing crashes or unpredictable behaviour. This script is directly connected with the functionality in the index.js that is imported and used to enhance the /data API endpoint. In index.js, the getStrongAuth function is invoked with a partitionKey derived from the request query parameters. The data comes from getStrongAuth is integrated with the aggregated response sent back to the client. Specifically, it gives details about strong authentication, such as whether it is included in a contract, or the number of authentications is in use.

This integration shows how the getStrongAuth function plays an important role for the overall application workflow. This helps to maintain modularity; the data retrieval logic is abstracted up to a reusable function, separating concerns. The index. js file is about managing the request and building the response, but the module is used to work with Azure Table Storage.

7.3 Connecting to Google Analytics API

The ga4. js module serves as the gateway to Google Analytics 4. It uses the @google-analytics/data library with a service account key to authenticate. It constructs queries dynamically to retrieve page view metrics for given time frames. This module extracts raw

analytics data and processes it to calculate derived metrics like total visits and changes from one month to the subsequent month.

```
1   const { BetaAnalyticsDataClient } = require('@google-analytics/data');
2   const { GoogleAuth } = require('google-auth-library');
3   const dotenv = require('dotenv');
4   dotenv.config();
5
6   const keyFilePath = process.env.GOOGLE_APPLICATION_CREDENTIALS;
7
8   const auth = new GoogleAuth({
9       keyFile: keyFilePath,
10      scopes: 'https://www.googleapis.com/auth/analytics.readonly',
11  });
12
13  const initializeAnalyticsDataClient = async () => {
14      const authClient = await auth.getClient();
15      return new BetaAnalyticsDataClient({ authClient });
16  };
17
18  const propertyId = process.env.PROPERTY_ID;
19
20  const getFirstAndLastDayOfMonth = (year, month) => {
21      const startDate = new Date(Date.UTC(year, month - 1, 1));
22      const endDate = new Date(Date.UTC(year, month, 0));
23      return {
24          startDate: startDate.toISOString().split('T')[0],
25          endDate: endDate.toISOString().split('T')[0],
26      };
27  };
28
29  const fetchGA4Data = async (stringFilterValue, startDate, endDate) => {
30      console.log(`[EXTERNAL CALL] GA4 - Fetching visits for Page Filter: ${stringFilterValue}, Date Range: ${startDate} to ${endDate}`);
31      const analyticsDataClient = await initializeAnalyticsDataClient();
32
    const request = {
        property: `properties/${propertyId}`,
        dateRanges: [{ startDate, endDate }],
        metrics: [{ name: 'screenPageViews' }],
        dimensions: [{ name: 'date' }],
        dimensionFilter: {
            filter: {
                fieldName: 'unifiedPagePathScreen',
                stringFilter: {
                    matchType: 'CONTAINS',
                    value: stringFilterValue,
                    caseSensitive: false,
                },
            },
        },
    };

    const [response] = await analyticsDataClient.runReport(request);

    if (!response || !response.rows || response.rows.length === 0) {
        throw new Error('No data available for the given dates');
    }


    return response;
};

exports.getVisits = async (stringFilterValue, month, year) => {
    const { startDate, endDate } = getFirstAndLastDayOfMonth(year, month);
    const response = await fetchGA4Data(stringFilterValue, startDate, endDate);

    const totalVisits = response.rows.reduce((acc, row) => acc + parseInt(row.metricValues[0].value, 10), 0);
    return totalVisits;
};
```

Figure 8. Connecting to Google Analytics

The first thing happens here in the script is to import dependencies. The @google-analytics/data package uses the BetaAnalyticsDataClient to query GA4 data through the Reporting API. The google-auth-library module contains GoogleAuth class for doing the authentication. The dotenv library is used to import environment variables, ensuring that the service account key file path and GA4 property ID are hardcoded in the application.

The keyFilePath property refers to the authentication service key file specified by the environment variable GOOGLE_APPLICATION_CREDENTIALS. Then, it sets up

GoogleAuth with this file and the https://www.googleapis.com/auth/analytics.readonly scope, providing least privilege read-only access for security.

The function initializeAnalyticsDataClient is defined to initialize the GA4 client. The first step in this asynchronous function is to get an authorized client using the GoogleAuth instance and then creates an instance of the BetaAnalyticsDataClient with the authenticated client. By keeping authentication initialization separate from the actual application logic, greater modularity in the code is ensured and makes the code easier to test and debug.

The other important part of the script is the utility function getFirstAndLastDayOfMonth. This function expects start date and end date of a given month and year. The dates are using UTC format to avoid timezone issues, and the function returns them in the ISO 8601 format (i.e. YYYY-MM-DD) which is compatible with the GA4 API.

The main functionality of the script is implemented in the fetchGA4Data function. This is an asynchronous function that queries the GA4 Reporting API for metrics related to screen page views for a date range and filter. The function first calls initializeAnalyticsDataClient to initialize the GA4 client. It then builds the request payload for the GA4 client's runReport method. This request specifies which property to query from, date ranges, metrics, dimensions, as well as a dimension filter.

The filter is important here, as it helps to narrow down the query to return only those data that have the unifiedPagePathScreen with the value specified by the stringFilterValue. The matchType is CONTAINS, and the filter is case insensitive. This enables the application to fetch data for all pages that match with the given filter value, regardless of the case.

This makes a call to the runReport method with the request and waits for the response. If the response is empty or does not have rows, an error is thrown due to not having any data for the corresponding date range. This means that downstream consumers of the data handle the case where there is no analytics data.

The getVisits function wraps the complete workflow. It receives stringFilterValue, month, and year as inputs. It first calls getFirstAndLastDayOfMonth to get the start and end dates for given month and year. Then it calls fetchGA4Data with the filter value and date range. The function handles the response by looping through the table rows in the result set and summing the values of the screenPageViews. This is the total visits for the given month by filter. It also fetches visit data from the previous month to compare percentage changes trend in traffic.

In index. js, the imported getVisits function is used to get the total number of visits for a given site and date ranges. The stringFilterValue came from the site details earlier in the workflow, while the month and year are being fetched from query parameters. The result comes from getVisits is then included in the accumulated response sent back to the client.

This module plays an important role to have a meaningful analytics data. This keeps the code modular and maintainable. This separation of concerns allows the application logic in index. js, to focus on the request handling and response construction, and letting the script to fetch and process the analytics data.

7.4 Connecting to Salesforce API

The salesforce.js module handles the Salesforce API integration, focusing on fetching account-related data using jsforce library. It queries through SOQL (Salesforce Object Query Language) for specific fields, active units, based on account names. By utilizing callback functions asynchronously, the module handles API responses without blocking other operations.

```
const jsforce = require('jsforce');

const conn = new jsforce.Connection({
    loginUrl: process.env.SF_LOGIN_URL
});

conn.login(process.env.SF_USERNAME, process.env.SF_PASSWORD + process.env.SF_TOKEN, (err, userInfo) => {
    if (err) {
        console.error("Login error:", err);
    } else {
        /*console.log("Successfully logged in to Salesforce");
        console.log("User Id:", userInfo.id);
        console.log("Org Id:", userInfo.organizationId);*/
    }
});

exports.getUnitsByName = async (name) => {
    return new Promise((resolve, reject) => {
        console.log(`[EXTERNAL CALL] Salesforce - Fetching units for Account Name: ${name}`);

        const query = `SELECT Name, Active_Units__c FROM Account WHERE Name = '${name}'`;

        conn.query(query, (err, result) => {
            if (err) {
                reject(err);
            } else {
                if (result.records.length > 0) {
                    resolve(result.records[0].Active_Units__c || 0);
                } else {
                    resolve(0);
                }
            }
        });
    });
};
```

Figure 9. Connecting to Salesforce

The script first imports the jsforce library to allow communication with Salesforce via its API. The jsforce.connection object makes connection with salesforce. The Connection constructor is initialized with the loginUrl that is dynamically loaded from the SF_LOGIN_URL

environment variable. This allows the connection to point to different Salesforce environments (production, Sandbox, etc) without the need to change the source code.

The conn.login method is used to authenticate the application with Salesforce. It requires the Salesforce username (SF_USERNAME), password (SF_PASSWORD), and security token (SF_TOKEN), those are securely loaded from environment variables. Salesforce's login mechanism needs both the password and token to be concatenated together. By using variables and modifying them in the process, sensitive credentials are not hard-coded, thus improving security and allowing flexibility to deploy across environments.

The login method takes a callback function which will be called after the authentication attempt. If an error occurs during login, the error is logged to the console, and the script exits the login process. Upon successful authentication, the script has access to the userInfo object with information about the authenticated user such as user ID, and organization ID.

After authenticating the connection, the script exports a function called getUnitsByName. This function is designed to query Salesforce data for a specific account name and fetch the value of the Active_Units__c field. The function follows a promise-based approach as Salesforce query operations are asynchronous in nature.

The getUnitsByName method constructs a SOQL (Salesforce Object Query Language) query. This query selects the Name and Active_Units__c fields from the Account object where the Name matches the provided value. The query is then executed using the conn.query method which sends the query to Salesforce and returns the result.

The callback function within conn.query processes the query result. If an error occurs while querying, the promise is rejected with the error object and the calling code can take appropriate action. After a successful query, the script inspects the result. If there are records in the result, it gets the Active_Units__c field from the first record and resolves the promise with its value. If the field is undefined or there are no records, the promise resolves with a default value of 0, ensuring that the function always returns a valid numeric value.

This function connects to the index.js file in the /data API endpoint. In index.js, the getUnitsByName function is imported and generate response with the name parameter, that comes from the query parameters of the incoming API request. Finally, the returned value, which indicates the count of active units associated with the specified account, is incorporated into the aggregated response that is sent back to the client..

The purpose of this script is simply to allow the applications to incorporate Salesforce data within its capabilities. This module isolates the Salesforce interaction, adhering to the separation of concerns principle. The main logic in index. js stays clean and only responsible for handling requests, while all the Salesforce-specific logic is encapsulated here.

7.5 Database Setup with PostgreSQL

The db. js module handles interaction with PostgreSQL database. Using the pg library, this module fetches site-related information from a PostgreSQL database. The code is designed to connect to the database securely, execute a query, and return the results. It uses Parameterized SQL Queries to directly access the data securely eliminating the risk of any SQL injections. The own module of backend ensures that any changes in database schema or query logic will not affect other parts of the system.

```
const { Client } = require('pg');

exports.fetchSiteDetails = async (siteId) => {
    const connectionString = process.env.DATABASE_URL;

    if (!connectionString) {
        throw new Error('Connection string is not defined');
    }

    const client = new Client({
        connectionString,
    });
    await client.connect();
    const res = await client.query('SELECT * FROM users WHERE site_id = $1', [siteId]);
    //console.log('Database Response:', res.rows); // Log the database response
    await client.end();
    return res.rows[0];
};
```

Figure 10. Database setup

The script imports the pg library, which includes the Client class for connecting to the PostgreSQL database. The fetchSiteDetails function is then exported as an async function to be used in other parts of the application. The function expects siteId as a parameter and queries the database for siteid related information.

The above function fetches the database connection string from DATABASE_URL environment variable before connecting to the database. This makes database credentials such as host, port, and database name are stored securely in an environment file instead of hardcoded in the source code. The DATABASE_URL is needed to configure properly otherwise it will stop further execution and security can be compromised.

After verifying the connection string, a new Client instance is initialized with the connection string. The Client is a single connection to a database created by connect method. This

function opens a connection to the database and allows the client to start executing queries. If the connection attempt fails, an error is thrown, which can be handled by the calling function.

After successfully connecting to the database, the function executes a query to retrieve site details. The query uses a parameterized SQL statement:

```
SELECT * FROM users WHERE site_id = $1
```

The $1 is the placeholder for the parameterized variable, and a query array is passed to the client with the siteid value using client.query method. This kind of parameterized query is a best practice to work with dynamic input and it secures database from SQL injection.

The client.query method runs the SQL statement and returns a promise resolving to the query result. The result will have a rows property which is an array of records with matching query criteria. Since, query gives the records filtered on site_id, the function will fetch its first record from the rows array using res.rows[0]. If there is no matched record the function will return undefined as response.

After fetching the result, the function calls client.end method to close the database connection, preventing resource leaks and ensuring efficient connection management in high-traffic applications. The function returns the first record from the query, containing site-specific details. This data, based on the users table structure, is used in other sections of the application including /data API endpoint in the index.js.

In the index. js file where this function is imported, used to get additional context from sites based on their siteId. For instance, in the /data endpoint, siteId is extracted from the request query parameters and the fetchSiteDetails function is called with that id. The resulting site details are then integrated into the response sent back to the client. This connection shows the contribution of the function to enrich API responses with database-driven information.

In summary, this module establishes a secure connection to a PostgreSQL database, performs a parameterized query to retrieve site-specific information, and returns the result. This is tightly coupled with the rest of the application, enriching API responses with relevant data. The process indicates following modern standards of database interaction with the use of environment variables, parameterized queries, and resource management.

7.6 Data processing and Aggregation

Request Handling and Data Aggregation process are the core component of the backends functionality. These features combine data from multiple sources efficiently and deliver it to

the frontend seamlessly. This process is managed within the /data endpoint, which is the main entry point for clients and is responsible for coordinating data retrieval and integration.

```
app.get('/data', async (req, res) => {
    try {
        const siteId = req.query.siteId;
        const partitionKey = req.query.PartitionKey || siteId;
        const name = req.query.name;
        const month = req.query.month;
        const year = req.query.year;

        if (!siteId || !name) {
            return res.status(400).send('siteId and name are required');
        }

        let siteDetails = {};
        let units = 0;
        let visits = 0;
        let previousMonthVisits = 0;
        let strongAuthIncludedToContract = 0;
        let strongAuthMade = 0;

        siteDetails = await fetchSiteDetails(siteId) || {};

        const stringFilterValue = siteDetails.stringfiltervalue;
        units = await salesforce.getUnitsByName(name);

        visits = await ga4.getVisits(stringFilterValue, month, year) || 0;
        const previousMonth = month === 1 ? 12 : month - 1;
        const previousYear = month === 1 ? year - 1 : year;
        previousMonthVisits = await ga4.getVisits(stringFilterValue, previousMonth, previousYear) || 0;

        const strongAuth = await azure.getStrongAuth(partitionKey) || [];
        if (strongAuth.length > 0) {
           strongAuthIncludedToContract = strongAuth[0].StrongAuthIncludedToContract;
            strongAuthMade = strongAuth[0].StrongAuthMade;
        }

        const visitsPerUnit = units ? (visits / units).toFixed(2) : null;
        const visitChange = previousMonthVisits ? ((visits - previousMonthVisits) / previousMonthVisits * 100).toFixed(2) : null;

        const data = {
            units,
            visits,
            visitsPerUnit,
            visitChange,
            strongAuthIncludedToContract,
            strongAuthMade,
            previousMonthVisits
        };

        res.json(data);
    } catch (error) {
        console.error('Internal Server Error:', error.message, error.stack);
        res.status(500).send('Internal Server Error');
    }
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Figure 11. Data processing & Aggregation

The server receives a request on the /data endpoint and checks if the query gets all the necessary parameters such as siteId, PartitionKey, name, month, and year to fetch records from the database. These parameters are essential for identifying and filtering the data relevant to the request.

Once the request is validated, backend executes an async process to make requests to different data sources. The db. js module to query PostgreSQL from the client-side to retrive in site-specific metadata, such as filter values. This data is extremely important to help subsequent queries to other services. The azure. js module retrieves partition-based data from Azure Table Storage, and the salesforce. js module sends a request to Salesforce's API to get account details like active units. Concurrently, the ga4. js module provides smooth interaction, passing data to Google Analytics 4 and captures user engagement data metrics such as visit counts for the given month and counts for the previous month.

These operations are asynchronous, allowing the server to remain responsive and capable of processing other requests concurrently. Once all data is retrieved, it is processed to compute derived metrics such as visits per unit and visit change percentages. These metrics are calculated by combining raw data, such as the total visits and active units, into meaningful insights that can be readily interpreted by the frontend.

Finally, the aggregated data is combined into a structured JSON response, which includes all requested information and computed metrics. This response is sent back to the client, providing a comprehensive and cohesive dataset for presentation or further analysis.

7.7 Error Handling and Logging

Error handling and logging are very important parts of the backend architecture that makes this application reliable, maintainable and transparent during the run time of the application. By implementing these mechanisms, application can efficiently identify potential issues that may come from data processing, external APIs, and even user requests.

Backend Error handling starts with the request validation phase. The /data endpoint verifies that necessary parameters (like siteId, PartitionKey, name, month, and year) have been provided. If any of the parameters are missing or invalid, the server replies with a 400 Bad Request status code and a message explaining the error message.

The backend uses try-catch blocks to wrap critical operations for operational errors, such as connectivity with external services or data mismatches. Each module—azure. js, db. js, ga4. js, and salesforce. js does have its own error handler and implements its own error handling logic specific to its data source. When such error occurs, the system logs the error message and stack trace in the error logging for debug. At the same time, the server responds to the client with a generic 500 Internal Server Error, as to not expose any sensitive internal information and safeguard system security.

7.8 Backend API Authentication with JWT

In this project, the need to protect backend APIs become a necessity. The main reason behind is the integration of various external sources. The application retrieved data from Google Analytics, Azure Table Storage, and Salesforce. In the initial design, the frontend application directly utilized API keys and keep the secrets into the environment variables. This was a very vulnerable approach as these variables were bundled into JavaScript that was publicly accessible. User could inspect the application using browser developer tools and extract sensitive credentials such as API keys or hardcoded tokens that are quite unacceptable.

Therefore, a secure mechanism was needed to authenticate frontend request without leaking the backend secrets to anyone. As discussed in Section 2.6, a secure secret (JWT_SECRET) is defined and stored within the backends' environment variables. The secrets never exposed to the frontend or any external users.

```
app.get('/get-token', (req, res) => {
    const JWT_SECRET = process.env.JWT_SECRET;

    const token = jwt.sign({ app: 'dashboard' }, JWT_SECRET, { expiresIn: '1d' });
    res.json({ token });
});
```

Figure 12. Backend API Authentication with JWT

The applications' backend has an API endpoint (/get-token) that generates JWT tokens and shares them with the frontend during runtime. The frontend fetches the short-lived token and keeps it in the application state temporarily. The token does not carry and sensitive information but identifies the requester as authorised. Protected API endpoints use middleware that verifies incoming tokens using the backend secret, and upcoming requests without valid tokens are denied with 401 Unauthorized or 403 forbidden responses.

```
const jwt = require('jsonwebtoken');
require('dotenv').config();

const JWT_SECRET = process.env.JWT_SECRET;

const authenticateToken = (req, res, next) => {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];

    if (!token) {
        return res.status(401).json({ error: 'Unauthorized: No token provided' });
    }

    jwt.verify(token, JWT_SECRET, (err, user) => {
        if (err) {
            return res.status(403).json({ error: 'Forbidden: Invalid token' });
        }

        req.user = user;
        next();
    });
};
```

Figure 13. Backend API Authentication with JWT

This solution significantly enhances the security of the application. The JWT secret, database credentials, Salesforce and Azure Storage keys explicitly remain on the server side. Unlike API keys, JWTs are generated at runtime and are not bundled into the frontend build. The short validity period of JWTs ensures that even if a token is compromised, it allows less time to misuse the system. Furthermore, JWTs are issued only for accessing specific API routes and do not grant broader access to backend infrastructure. This approach balances security with practicality. It avoids the complexity of full user authentication while ensuring that only trusted frontend clients can access protected backend resources.

By replacing hardcoded secrets and API keys with dynamically issued JWT tokens, the application’s security posture has been greatly improved. Now sensitive backend secrets are fully protected, and frontend tokens are now short-lived and securely transmitted. This solution effectively prevents unauthorized API access and ensures data privacy, while maintaining a simple, flawless frontend architecture that meets the requirements of the project.

8 Front-end Implementation

The front-end implementation is a React-based application that uses Azure authentication and retrieves data from Azure Table Storage and makes calls to a backend API to obtain customer analytics. The overall workflow involves authentication, fetching and processing data, rendering UI components, and ensuring a smooth user experience. React's component-based architecture aligns well with the project's need, ensuring that the application remains scalable and easy to manage. The project is initialized with npx create-react-app, a widely adopted tool for setting up a modern React development environment, offering a structured configuration that simplifies the development process.

The project structure follows a well-organized folder hierarchy, as demonstrated below:

```
client
├── src
│   ├── components
│   │   ├── Comparison.js
│   │   ├── ExportToPDF.js
│   │   └── TableData.js
│   ├── App.js
│   ├── App.css
│   └── index.js
├── public
│   └── index.html
├── .env
├── package.json
└── README.md
```

Figure 14. Frontend folder structure

The project structure follows a well-organized folder hierarchy that promotes a clear separation of concerns. Inside the client directory, the core of the frontend application located in the src folder. In the src folder, all the essential components, services, and configuration files are located. These files are collectively making the user interface and interactions. The components folder contains reusable components those construct the user interface including comparison functionality, PDF exports, login management, and tabular data presentation to ensure an interactive and scalable user interface. In addition, the services folder integrates utility functions for authentication, data retrieval, and the communication with external storage

systems and APIs. This approach ensures that the frontend implementation remain efficient, modular, and flexible to any future expansion with modern web application development.

8.1 Connecting the Frontend to the Backend

Connection between the frontend and backend is essential for continuous data flow. This connection enables user input, dynamic content updates, and real-time interaction within the application. The React-based frontend, communicates with the node.js and Express.js based backend through HTTP GET requests, utilizing Axios library for data exchange. This enables the application to dynamically fetch, and update customer analytics based on user interactions. Sensitive configuration like backend url are stored in environment variables (process.env.React_APP_BACKEND_URL) to ensure security and flexibility.

Data retrieval and state management take place within frontend's App.js file. The query parameters (siteId and name) handle query-based API requests and dynamically updates the UI. Personalized data is fetched from the URL using the useQuery() function, allowing the application to build a backend request based on the user's selection as follows:

```
const useQuery = () => {
    return new URLSearchParams(useLocation().search);
};
const query = useQuery();
const siteId = query.get('siteId');
const name = query.get('name');
```

Figure 15. Building query parameters

As it is shown in the Picture that the data retrieval process begins by fetching query parameters (siteId and name) from the URL, which determines the dataset to be fetched. The function useQuery() within App.js captures these parameters and dynamically builds an API request URL based on the user's input.

The fetchData() function makes an async HTTP GET request to the backend with an additional API key header to authorize access. The request fetches the customer analytics, passing the site id, customer name and the time period selected as parameters. It uses Axios to handle API calls that dynamically adjusts the displayed information based on the selected date range. The function is wrapped in the useCallback() hook to enhance efficiency and prevent unnecessary re-fetching. The process ensures that the function executes the

associated API calls only whenever the dependencies like date or selection of dataset change. Implementation is done as the code below.

```
const fetchData = useCallback(async (month, year) => {
    if (siteId && name) {
        const encodedName = encodeURIComponent(name);
        let url = `${process.env.REACT_APP_BACKEND_URL}/data?siteId=${siteId}&name=${encodedName}&month=${month}&year=${year}`;

        try {
            const response = await axios.get(url, {
                headers: {
                    'x-api-key': process.env.REACT_APP_API_KEY,
                },
            });
            setData(response.data);
        } catch (error) {
            console.error('Error fetching data:', error);
        }
    }
}, [siteId, name]);
```

Figure 16. Fetching customer analytics

This code shows how the application communicates with an external backend service and retrieves the customer data. Authentication header ensures that only authorized frontend that is configured with shared secret key, can communicate with the backend. It can communicate from even out of the browser scope. Once the data is successfully fetched, it is passed to the TableData component for rendering.

The useEffect() hook is used to fetch the data here at the very first mount of the component and also when dependencies (the selected date in this case) changes, keeping the rendered data more relevant and up-to-date. As shown in the following code snippet, the useEffect hook is used to make an initial API call when the component is mounted:

```
useEffect(() => {
    const now = new Date();
    const currentMonthName = now.toLocaleString('fi', { month: 'long' });
    setMonthName(currentMonthName.charAt(0).toUpperCase() + currentMonthName.slice(1));
    fetchData(now.getMonth() + 1, now.getFullYear());
}, [fetchData]);
```

Figure 17. useEffect hook for API call

When data is fetched from the backend, it is saved in the component's state using useState(), triggering a re-render of core UI components such as TableData.js and Comparison.js. Any kind of modifications in the backend data are immediately reflected on the frontend by state-driven rendering approach. This maintains consistency between the UI and the stored records.

On the backend side (index.js), the application ensure that only authorized frontend clients can access the /data route by validating a custom API key sent in the request headers.

```
app.use((req, res, next) => {
    const key = req.headers['x-api-key'];
    if (!key || key !== API_KEY) {
        return res.status(403).json({ error: 'Forbidden: Invalid API Key' });
    }
    next();
});
```

Figure 18. Authorized clients are only accessible here

The key-based protection ensures that without proper key, any requests to the backend are blocked. This implementation ensures an effective security layer without making complex authentication protocol. By using useState hook and conditional rendering (if (!data) return <di>Loading...</div>), frontend passed fetched data to TableData component for visualization.

The connection between the frontend and backend is a most important part of this project, what is designed to provide efficient, secure, and real-time data access. Axios for API requests, React hooks for managing side effects, Azure services for authentication and data storage, the application ensures seamless integration between the client and server. This setup enables dynamic data updates through fetching query parameter, state-driven rendering, and optimized API calls. Error-handling strategies, authentication enforcement, and partition-based filtering further enhance the system's reliability, security, and performance. This structured approach ensures users receive relevant up to date data on, improving the overall utility and effectiveness of the dashboard for better decision-making.

8.2 Handling User Interactions

The React-based frontend facilitates dynamic user interactions, enabling seamless data retrieval, authentication, and visualization in this application. In addition to authentication, the application allows the users to fetch dynamic data. The App.js component plays a core role in managing query-based data fetching by capturing URL parameters (siteId and name).

Axios library is used in the application to request data from the backend API. To get filtered data for any specific month and year, the application integrates calendar-based component (react-datepicker). When the user picks a new date, a callback function handleDateChange is invoked and the component updates the state of the application. A new API call is triggered to fetch data eventually, which is then rendered on the screen. React hooks (useEffect and useCallback) are used for data fetching optimization (avoiding unnecessary calling of APIs and avoiding updates if it’s not necessary).

```
const handleDateChange = date => {
    setSelectedDate(date);
    const selectedMonthName = date.toLocaleString('fi', { month: 'long' });
    setMonthName(selectedMonthName.charAt(0).toUpperCase() + selectedMonthName.slice(1));
    fetchData(date.getMonth() + 1, date.getFullYear());
};
```

Figure 19: Dynamic date based data search

App.js component has another notable feature as the DatePicker component, which provide users with the ability to choose a specific month and year to filter the displayed data. New date selection triggers a new API request, ensuring that the displayed data is always up to date. Implementation is done in the code below:

```
<DatePicker
    selected={selectedDate}
    onChange={handleDateChange}
    dateFormat="MM/yyyy"
    showMonthYearPicker
/>
```

Figure 20. User Interactions with Month/Year

Another type of interaction occurs through URL query parameters. This interaction enables users or external systems to directly access the dashboard with predefined siteId and name values. The useQuery() hook, previously discussed in Section 8.1, extracts these values to dynamically construct the API request via the fetchData() function.This approach of the application doesn't need direct user intervention to load relevant analytics automatically.

Furthermore, the system incorporates conditional rendering to enhance usability. While the data is being loaded asynchronously, a loading message is shown, improving the user experience:

```
if (!data) {
    return <div>Loading...</div>;
}
```

Overall, this interaction model highlights the project's focus on usability, responsiveness, and dynamic behaviour. By combining controlled components, state management, and query-

based routing, the application offers a user-centric experience where input directly governs the backend data flow and frontend rendering logic.

8.3 Creating the Dashboard UI

The dashboard UI is the main visual component of the application that allows users to interactively analyse key customer metrics in a tabular format. It is designed using React's component-based architecture. The dashboard fetches and displays data on business units, website visits, authentication statistics, and changes in user engagement over time.

The TableData.js component is responsible for rendering the dashboard table with the connection of Comparison.js, and ExportToPDF.js. The TableData.js component is built to dynamically display customer data passed as props. The dashboard provides a smooth user experience with its dynamic state management, API-based data retrieval, and responsive UI components. Below is the implementation of TableData.js:

```
import React from 'react';
import '../App.css';

const TableData = ({ data }) => {
    const { units, visits, visitsPerUnit, visitChange, strongAuthIncludedToContract, strongAuthMade, previousMonthVisits } = data;

    return (
        <div className="container">
        <h1>Customer Dashboard</h1>
            <table border="1" cellPadding="10" cellSpacing="0">
                <thead>
                    <tr>
                        <th>Units</th>
                        <th>Visits</th>
                        <th>Visits/Units</th>
                        <th>Visits Last Month</th>
                        <th>Visit Change Compared to Last Month</th>
                        <th>StrongAuth Included To Contract</th>
                        <th>StrongAuth Made</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td>{units || 'N/A'}</td>
                        <td>{visits || 'N/A'}</td>
                        <td>{visitsPerUnit || 'N/A'}</td>
                        <td>{previousMonthVisits || 'N/A'}</td>
                        <td>{visitChange ? `${visitChange}%` : 'N/A'}</td>
                        <td>{strongAuthIncludedToContract || 'N/A'}</td>
                        <td>{strongAuthMade || 'N/A'}</td>
                    </tr>
                </tbody>
            </table>
        </div>
    );
};
export default TableData;
```

Figure 21. Dashboard UI Implementation

Main part of the dashboard implementation is done the TableData.js component, which renders a structured table displaying key performance indicators. The component handles dynamic data which is passed to it as props and gives statistics about the customers in real-

time by showing business units, website visits, visits per unit, previous month visits, authentication statistics, and visit change percentages. The table ensures data consistency by replacing undefined or missing values with the placeholder 'N/A' and uses a standard HTML table elements with clearly defined headers and cell spacing to enhance readability.

The Comparison.js component improves the dashboard's analytical capability by calculating the percentage change in website visits between the current and previous month. This functionality enables the application to track month-over-month performance trends. It ensures accurate calculations by defaulting the change percentage to zero if previous month data is unavailable. This metric helps users to track trends, identify growth or decline, and make data-driven adjustments.

Apart from data visualization, the dashboard also contains functional elements that improve user interaction and usability. DatePicker component within App.js implements the date selection feature, allows users to filter the displayed data based on a chosen month and year. The application dynamically updates the dataset when a user selects a new date, making sure that user always have most relevant data access. The ExportToPDF.js component displayed dashboard data as PDF document which allows the users to generate reports for offline access or sharing.

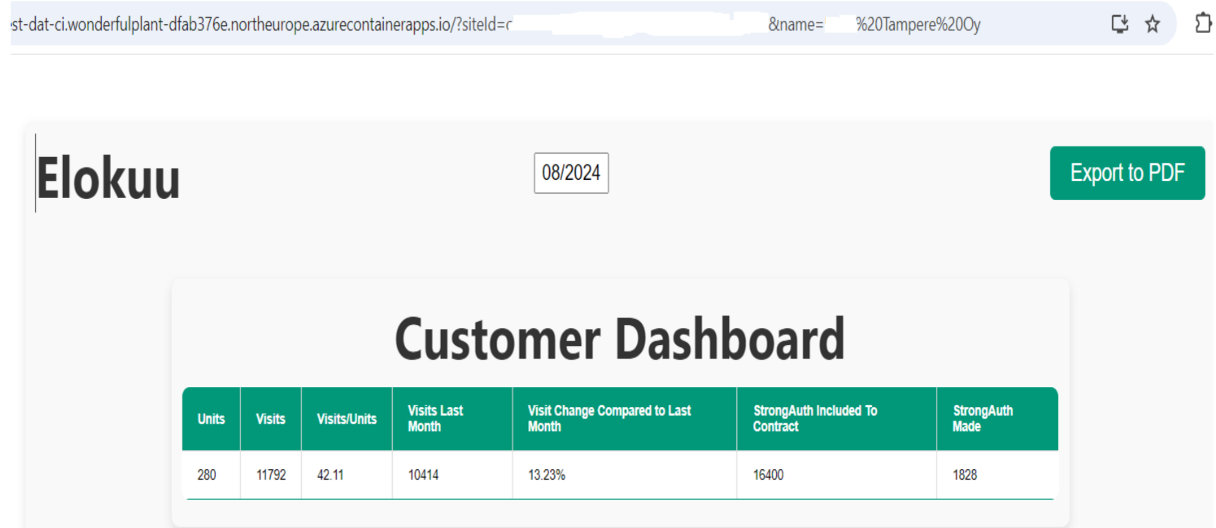

Figure 22: UI View Dynamic screen

Elokuu 08/2024 Export to PDF

Asiakas	Taloyhtiöt	Käynnit	Suhde	Käynnit edellisessä kuussa	Muutos edelliseen kuukauteen	Sopimukseen kuuluvat tunnistautumiset	Tehdyt tunnistautumiset
Jyväskylä	141	5670	40.21	6080	-6.74%	7900	930
Porvoo	117	7070	60.43	6871	2.90%	6600	821
Tampere	280	11792	42.11	10414	13.23%	16400	1828
Imatra	62	468	7.55	427	9.60%	3400	120
Hämeenlinna	162	5410	33.40	6993	-22.64%	9400	899
Lahti	216	8970	41.53	9646	-7.01%	12400	1273
Mikkeli	96	2320	24.17	1982	17.05%	5800	362
Lappeenranta	93	2332	25.08	3329	-29.95%	5400	424
Joensuu	81	1678	20.72	1785	-5.99%	4600	314
Oulu	190	4367	22.98	4661	-6.31%	11000	916
Kymi	231	13896	60.16	11654	19.24%	13300	687

Figure 23. UI View Static view

In summary, the dashboard UI provides a powerful tool for analysing customer engagement, delivering an interactive, data-driven and result-oriented experience. The dashboard enhances usability, accessibility, and decision-making capabilities through structured tabular representation, comparative analytics, date-based filtering, and export functionalities. Taking full advantage of React's state-driven architecture and render-on-demand capabilities, the application provides a responsive, user-centric interface for better business insights.

8.4 Implementing Data Visualizations

Data visualization is a key factor in improving the interpretability of customer analytics by converting raw numerical data into meaningful insights. In this application, visualization does comparative analysis of website visits to visualize trends and differences in customer engagement over time. The Comparison. js component is the core visualization component that calculates and shows the output as a percent change in visits for current vs last month. By implementing real-time analytics within the dashboard, the visualization component ensures that users can easily assess fluctuations in customer interactions trends, enabling effective decision-making based.

The visualization logic within Comparison.js is designed to dynamically calculate the percentage change in website visits. It is calculated using a formula that calculates the relative difference between the number of visits in the current month and the previous month:

```
client > src > components > JS Comparison.js > ...
1   import React from 'react';
2
3   const Comparison = ({ currentMonthViews, previousMonthViews }) => {
4       const visitChange = previousMonthViews ? ((currentMonthViews - previousMonthViews) / previousMonthViews) * 100 : 0;
5
6       return (
7           <div>
8               <h2>Current Month Views: {currentMonthViews}</h2>
9               <h2>Previous Month Views: {previousMonthViews}</h2>
10              <h2>Visit Change Compared to Last Month: {visitChange.toFixed(2)}%</h2>
11          </div>
12      );
13  };
14
15  export default Comparison;
```

Figure 24. Data visualization implementation

```
The visitChange calculation uses the formula:

visitChange=((currentMonthViews - previousMonthViews)/previousMonthViews) * 100
```

The calculation ensures that if there is no data available for the previous month, the percentage change can be computed as default zero, thereby preventing errors caused by division by zero. The percentage is then formatted to two decimal places and displayed on the UI, giving users a clear and concise insight of visit trends.

Beyond numerical computations, the dashboard ensures dynamic data visualization according to user inputs. User can select filter data by month and year through App.js, which directly impacts visualization component. In response to the new date selection, the system updates the dataset, prompting the Comparison.js component to recalculate and display the updated visit change percentage to ensure current and real-time dataset in the visualization module.

Along with generating visualizations on screen, users can export visualized data to be generated as a PDF report via the ExportToPDF component. This is a unique functionality which allows the users to save the analysis and share the results in a standard format which made the data driven analysis to be applicable beyond the data visualization application.

9 Deployment

The deployment phase is a critical part of this application development lifecycle, where the application is made accessible to the end users. This section focuses on Containerization with Docker and deploying applications with Azure Container Apps and Azure Container Registry. The process allows the application to scale and manage containers easily according to need.

9.1 Containerization with Docker

The application is Dockerized to ensure consistent deployment in different environments. Docker makes a package for the application and its dependencies into a lightweight, portable container that can make it run consistently in various platforms. In the initial step, authentication with Docker Desktop has taken place to access the container registry securely. Azure ACR requires this authentication step to get the pushed images. Dockerization provides better scaling and consistency between development, and production environments.

9.2 Creating the Dockerfile

In this project, a Dockerfile is created to package the data reporting application into a Docker image. This image ensures that all necessary components are included to run the application. Two separate Dockerfiles are created in this application to containerize the frontend (React) and backend (Node.js) components. The Docker images utilize multi-stage builds and efficient dependency management to optimize the final image sizes.

Frontend: React Application Dockerfile

The frontend Dockerfile is using multi-stage capabilities to separate build and runtime environments for performance and security reasons.

Stage 1: Build the React Application

```
FROM node:18.16.1 AS build

WORKDIR /usr/src/app

COPY package*.json ./
RUN npm install

COPY . .
RUN npm run build
```

Figure 26. Docker Image Build

The implementation uses the base image node:18.16.1 that provides a Node.js runtime environment. The WORKDIR directive defines the working directory as /usr/src/app. COPY package*. json. COPY / copies the package files to the container to install dependencies with npm install. The source code of the application is then copied with COPY.., and afterwards the react application is built with npm run build.

Stage 2: Serve the Application Using a Static Server

```
FROM node:18.16.1

WORKDIR /usr/src/app
RUN npm install -g serve

COPY --from=build /usr/src/app/build ./build

EXPOSE 3000
CMD ["serve", "-s", "build", "-l", "3000"]
```

Figure 27. Docker image stage 2

The node:18.16.1 base image creates a lightweight runtime environment. It is a static file server that installed globally to serve the React build files. From the Stage 1, the build artifact is copied into the new image with COPY --from=build /usr/src/app/build. /build. The container exposes port 3000 and executes the serve command to run the application.

Backend: Node.js Application Dockerfile

The Dockerfile for the backend specifically builds and launches a Node. js server that deals with API request and contains business logic.

```
FROM node:18.16.1

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install
COPY . .
COPY .env .env

EXPOSE 3002
CMD ["node", "index.js"]
```

Figure 28. Docker image for backend

The base image node:18.16.1 ensures that it works with Node. js features and libraries. The working directory is set to /usr/src/app as per the front-end structure. The package files are copied and install dependencies with npm install. The source code and environment configuration file. env are passed to the container. Port 3002 is exposed to allow external access to the API server. The application initiated with node index. Js, that starts up the backend server.

This containerization approach using Docker enhances the application's portability, which ensures that the application runs consistently regardless of where it is hosted and how it is run. The frontend uses a multi-stage build to minimize the final image size, while the backend configures for the simplest and fastest Node.js-based services.

9.3 Building Docker Images

After creating Dockerfile, building Docker image comes as the next step. Docker image is a software package that is lightweight and includes everything needed to run the software, including the code, runtime, libraries, and environment variables. A Dockerfile is used to create the image, which is a series of commands that specify the environment and processes required to run the application. Docker images are built using a simple process with the docker build command as follows:

```
docker build -t myacr.azurecr.io/my-server-image:1.0 -f Dockerfile .

docker build -t myacr.azurecr.io/my-client-image:1.0 -f Dockerfile .
```

Figure 29. Docker commands for Build images

In this command, two images are created, one is for backend and other one is for frontend of the application. Docker build initiates the image creation process, while the -t flag assigns a specific tag, myacr.azurecr.io/server:1.0, which specifies the repository location in Azure Container Registry (ACR) and the version of the image. The -f Dockerfile option specifies the Dockerfile to be used for the build, and the period (.) at the end indicates that the Dockerfile is in the current directory. Both images undergo within the same processes. When the Docker image is successfully built, the image is pushed to Azure Container Registry (ACR) to store it securely and deploy it easily across various environments.

10 Deployment on Azure Container Apps

The application is deployed into Azure Container Apps that enables developers to launch and run any containerized application without the need of provisioning infrastructure. The deployment process on Azure Container Apps involves several steps, including setting up an Azure Container Registry, pushing Docker images, configuring Azure Container Apps, and deploying the application.

10.1 Setting Up Azure Container Registry

Azure Container Registry (ACR) is a type of managed Docker registry service provided by Microsoft Azure. It enables to keep and manage container images in a secure manner. Commissioning company has Azure pay as you go subscription for the resources management. Azure container registry is created using Azure portal which is used to store locally created docker images. This step is crucial as it allows for secure storage and access to the images that is needed for deployment with Azure services. Container Registry is created by the following steps that is quite similar as Container Apps.

Home >

Create container registry ...

Basics Networking Encryption Tags Review + create

Azure Container Registry allows you to build, store, and manage container images and artifacts in a private registry for all types of container deployments. Use Azure container registries with your existing container development and deployment pipelines. Use Azure Container Registry Tasks to build container images in Azure on-demand, or automate builds triggered by source code updates, updates to a container's base image, or timers. Learn more

Project details

Subscription * Pay-As-You-Go

Resource group *

Create new

Instance details

Registry name * Enter the name

.azurecr.io

Location * North Europe

Use availability zones

Availability zones are activated on premium registries and in regions that support availability zones. Learn more

Pricing plan * Standard

Review + create < Previous Next: Networking >

Figure 30. Creating Azure Container Registry

After setting up the ACR, Docker images are pushed to registry with the following commands:

```
docker push myacr.azurecr.io/my-server-image:1.0

docker push myacr.azurecr.io/my-client-image:1.0
```

Figure 31. Docker commands for push images

This command uploads the tagged image to the specified repository in ACR, allowing for efficient distribution, version control, and deployment in both development and production environments. The command pushes the image to the container registry. When the image is pushed, it becomes available for deployment on Azure services.

10.2 Configuring Azure Container Apps

The deployment process for both the frontend and backend applications are carried out using Azure Container Apps via the Azure Portal. There are some configurations steps that is followed steps to ensure the seamless deployment of containerized applications. The process starts with the selection of basic project settings and progressing through container image specification, network configuration, and final validation.

The deployment starts by navigating to the Container Apps section in the Azure Portal to create a new container app. In the Basics tab, the subscription is set to Pay-As-You-Go to manage resource costs effectively. The existing resource group is selected for easier management and monitoring. Each container app has its own unique name, allows to separate the frontend and backend service. The source of the deployment is set to Container image, which indicates that the application will deploy using the pre-built Docker images stored in the Azure Container Registry (ACR). The region is specified as East Europe to optimize performance, and the resource group environment is selected to capture both applications within the same secure boundary.

Home > Container Apps >

Create Container app ...

Passed

Basics Container Ingress Tags **Review + create**

Project details

Subscription	Pay-As-You-Go
Resource group	analytics171
Name	unified-app-server

Container Apps Environment

Region	northeurope
Container Apps Environment	managedEnvironment-analytics171-8b4e

Container

Name	unified-app-server
Image source	Azure Container Registry
Registry	analytics172.azurecr.io
Image	unified-app-server
Image tag	1.0
Command	
Arguments	
Number of CPU cores	0.25
Memory size (Gi)	0.5
Ingress settings	Accepting traffic from anywhere
Ingress type	HTTP
Transport	Auto
Insecure connections	Not allowed
Port	3002

Create < Previous Next Download a template for automation

Figure 32. Creating container Apps

After basic configuration, the process moves to Container tab where the Docker images are configured. The images from ACR are used for the frontend as well as the backend. The resources are allocated as default to 0.25 CPU cores, 0.5 Gi memory for each of the application by keeping the opportunity to scale later if need arises.

Ingress of applications is configured in Ingress tab with external access. The frontend and backend are set to be exposed publicly by the external ingress. The ports are indicated to match the ones revealed in the Dockerfiles: port 3000 for the frontend React app and port 3002 for the backend Node.js API. This is done so both applications could be accessed externally and also had secure communication channels established as well.

The Tags tab is used to to add metadata for easier resource management. Tags are assigned to classify the applications and their environment, organizing and tracking costs, and grouping resources within Azure. Nothing is configured in Tag section for this application deployment.

Finally, in the Review + Create tab, all the configurations are reviewed for accuracy. Azure's automated validation process assures that there are not any misconfigurations or missing parameters. After validating successfully, the Create button is selected to finalize the deployment. Azure Container Apps then takes care of provisioning resources and deploying the containerized applications.

The same process is repeated twice, once for deploying the backend application and once more for the frontend application. Both apps are successfully deployed in a cloud environment, utilizing the power of Azure to handle scaling, load balancing, secure connectivity, without manual intervention.

10.3 Deploying the Application on Azure

Once the Azure Container App is configured, the application is deployed using the Azure portal. The deployment phase involves starting the container instance, ensuring the application is running and accessible. During the deployment phase, logging and monitoring are established to confirm that the application is functioning expectedly.

Deployment process involves multiple steps which includes containerizing the application & finally deploying it in Azure Container Apps. By following best practices for containerization and deployment, the application ensures its scalability and security in the cloud environment.

11 Testing Implementation & Performance Measurement

Testing and validation are implemented in this project to ensure the correctness, stability, and security of both the frontend and backend components. To verify accurate rendering of React components and the reliability of data-fetching logic from API, Jest Unit testing and React testing library are being used in this project. Supertest and Jest are used in the backend to test Express.js API endpoints. This test ensures that incoming responses matched with the expected output and invalid requests are properly negotiated. Integration testing validated the exchange between the application components with the external services. It confirms that end-to-end data flow is operated smoothly with the cooperation of mocking. Additionally, JWT is implemented to verify that unauthorized access to API routes is restricted in the runtime. This multi-layer method provides confidence on application's functionality.

```
//Testing Scripts (TableData.test.js)
import { render, screen } from '@testing-library/react';
import TableData from './TableData';

test('renders TableData with headings', () => {
  const dummyData = {
    units: 5,
    visits: 20,
    visitsPerUnit: 4,
    visitChange: 10,
    previousMonthVisits: 15,
    strongAuthIncludedToContract: 1,
    strongAuthMade: 1,
  };

  render(<TableData data={dummyData} />);
  expect(screen.getByText('CustomerDashboard')).toBeInTheDocument();
  expect(screen.getByText('Units')).toBeInTheDocument();
  expect(screen.getByText('5')).toBeInTheDocument();
});

//App.test.js
import { render, screen } from '@testing-library/react';
import WrappedApp from './App';

test('shows loading screen initially', () => {
  render(<WrappedApp />);
  expect(screen.getByText(/loading/i)).toBeInTheDocument();
});
```

Figure 33. Testing Scripts

As it is shown in the code snippet, the TableData component is tested with dummy props to verify that headings like unit and data points are rendered correctly. The App component is tested to ensure the loading screen appears before content loads.

In the Frontend, comprehensive unit and integration tests are developed using Jest and React Testing Library to ensure the stability and readiness of key UI components. The test case successfully passed all defined segments. Overall, the implemented tests validate the structural integrity of the interface and ensure reliable behaviour of essential components.

```
PASS  src/components/Login.test.js
PASS  src/components/TableData.test.js
PASS  src/App.test.js
```

```
Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.859 s, estimated 3 s
```

Figure 34. Front-end Test Outcome

With Mocked data, the TableData component is independently tested to verify that it renders all expected headings and data points, such as "Customer Dashboard" and "Units". Simultaneously, the App component testing result confirms that it initially displays a loading message and subsequently renders the dashboard content after successful data retrieval. To do isolated testing without depending on real API endpoints or browser-specific features, several external dependencies including axios, jspdf, and html2canvas are being mocked. This approach ensured that asynchronous data fetching, and PDF generation functionalities did not interfere with test execution in the JSDOM environment.

In the Backend, to ensure the reliability of the backend system, a unit and integration test is implemented using Jest and Supertest for the /data API endpoint. This test verifies that the endpoint correctly handles HTTP GET requests when provided with valid query parameters, such as siteId, name, month, and year.

```
PASS tests/app.test.js
  GET /data
    √ should respond with JSON containing expected fields (40 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.836 s, estimated 7 s
Ran all test suites.
```

```
PASS tests/app.test.js (6.348 s)
      at index.js:49:28

PASS tests/app.test.js (6.348 s)

PASS tests/app.test.js (6.348 s)
  GET /data
    √ should respond with JSON containing expected fields (4589 ms)
PASS tests/app.test.js (6.348 s)
  GET /data
    √ should respond with JSON containing expected fields (4589 ms)
  GET /data
    √ should respond with JSON containing expected fields (4589 ms)

    √ should respond with JSON containing expected fields (4589 ms)
```

Figure 35. Backend Test outcome

The test follows a client request and inspects the response. The test confirms that it returns the response as expected structure and keys in JSON format. External services such as Azure, Google Analytics (GA4), and Salesforce provide dynamic data. These are mocked to ensure test isolation and avoid reliance on live cloud resources. The successful execution of the test confirms that the backend can integrate and respond with computed metrics—like units, visits, visitChange, and strongAuthMade—even in a simulated environment. This type of backend testing validates both the application logic and API contract, contributing to the stability and reliability of the system.

11.1 Performance Measurement

In this project, two types of method are used to see the speed, error rate, and efficiency measuring. Azure monitor shows the high-level operational metrics from the production, but the application-level measurements give better insights about speed and error count per request. A well-structured logging mechanism is implemented that captures critical runtime metrics are later analysed to assess three key factors speed, error rate, and efficiency.

11.1.1 Application-level

Speed is measured through middleware that logged response times for each API request. The middleware calculates the duration between receiving a request and sending a response. These logs, recorded in s standardized format, enabled calculation of minimum, average, and maximum response times of backend under different load conditions. Implementation below:

```
//API Timing Logs in backend
app.use((req, res, next) => {
    const start = Date.now();
    res.on('finish', () => {
        const duration = Date.now() - start;
        console.log(`Request to ${req.originalUrl} took ${duration}ms`);
    });
    next();
});

//Application level login
app.use((err, req, res, next) => {
    console.error(`Error during request to ${req.originalUrl}:`, err.message);
    next(err);
});

//External calls logging → Salesforce, Azure, GA4
console.log(`[EXTERNAL CALL] Azure Table Storage - PartitionKey: ${partitionKey}`);
console.log(`[EXTERNAL CALL] Salesforce - Fetching units for name: ${name}`);
console.log(`[EXTERNAL CALL] GA4 - Fetching visits for: ${stringFilterValue}, ${month}/${year}`);
```

Error rate is assessed by analysing HTTP status codes and error logs generated during request processing. Any requests resulting in status code of 500 or captured by the error handling middleware are counted as failures. Metric captured as successful run is called efficiency and that is achieved outbound requests to the external services.

Multiple API request sessions are conducted below. These are the integration of backend logging, to analyse speed, error rate, and efficiency.

```
Server is running on http://localhost:3002
[METRICS] GET /get-token - Status: 200 - Time: 12ms
[METRICS] GET /get-token - Status: 304 - Time: 2ms
[METRICS] GET /get-token - Status: 200 - Time: 3ms
[METRICS] GET /get-token - Status: 304 - Time: 2ms
```

Endpoint	Status	Response Time
/get-token	200	12ms
/get-token	304	2ms
/data	200	4839ms

```
[EXTERNAL CALL] GA4 - Fetching visits...
[EXTERNAL CALL] GA4 - Fetching visits...
[EXTERNAL CALL] Azure Table Storage - PartitionKey: ...

[METRICS] GET /data?siteId=69&name=GEIM&month=5&year=2025-Status:200-Time:4839ms
```

Speed analysis presents that under a general condition, requests completed rapidly. Most of the requests are completed within 2 to 12 milliseconds. In case of external service calls, request processed time reached to nearly 5 seconds. This is quite expected that third-party services take more times, but it remained within the acceptive boundary for a data dashboard.

Error rate evaluation showed that the backend is highly reliable. Across all tests sessions, no major errors are identified in the system. In the test scenario, an intentional failure is made, therefore units are missing from Salesforce. Efficiency analysis indicating that each API request triggered an average of three external service calls to GA4 and Azure Table storage. Overall, the results validated that the backend is reliable, efficient and ready for production.

11.1.2 Production-level

According to telemetry captured from Azure Monitor, the application achieves a very low average load time that are shown in the following images:

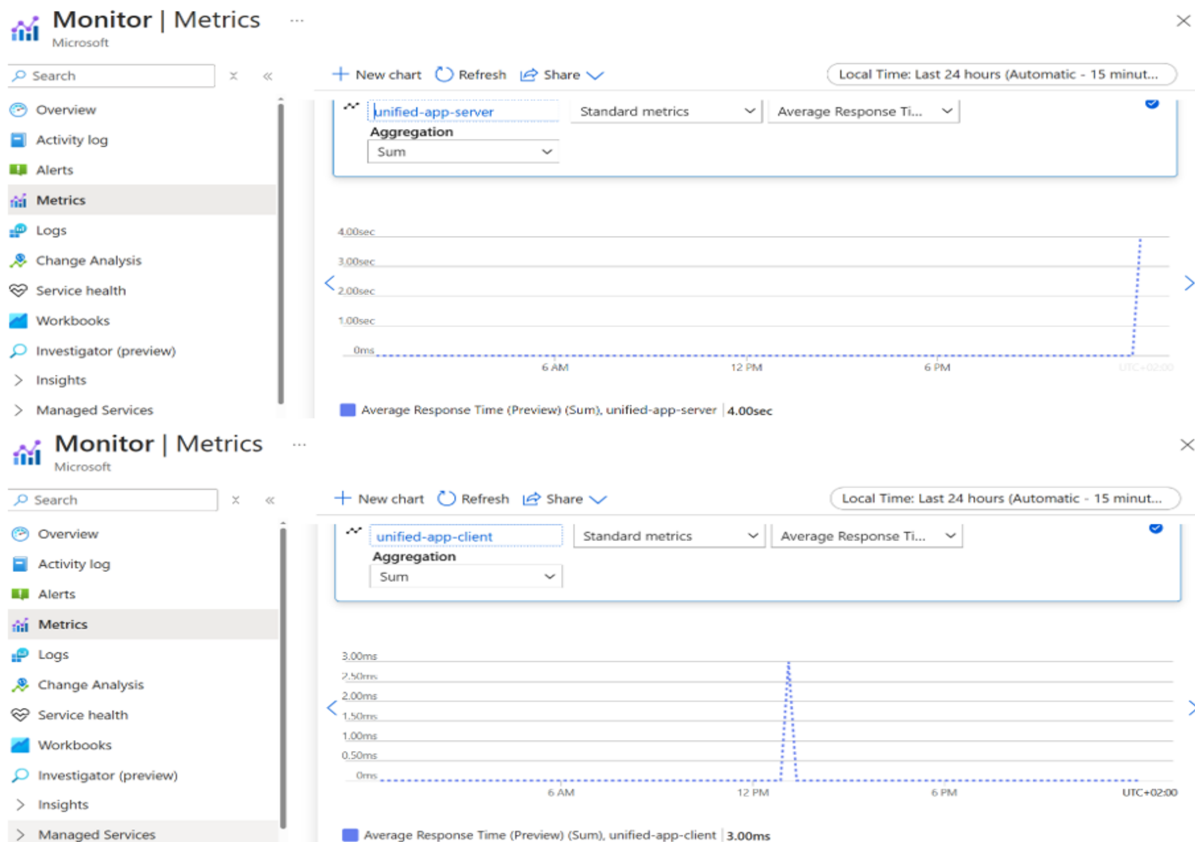

Figure 18. App metrics from Azure Monitor

The Azure Metrics screenshots display the average response time for two deployed components. The backend (unified-app-server) and the frontend (unified-app-client), both hosted using Azure Container Apps and Azure Container Registry. The metric tracked here is Average Response Time (Preview), which measures the time taken to process HTTP requests. The backend, labelled as unified-app-server, shows a consistently low average response time of 4.00ms, while the frontend unified-app-client has even lower average response time of 3.00ms indicating fast and efficient processing of API requests.

However, the analysis is conducted based on the limited number of samples under controlled testing conditions. To get more reliable understanding on performance, larger dataset needs to be included in the future testing. Increased data volume, higher user appearances, and broader data ranges could further impact the data response times. To ensure sustainable efficiency, various optimization strategies could be considered in the application. Batching requests to reduce number of external service calls and schedule background jobs are notable options to maintain acceptable performance level while meeting user expectations.

12 Evaluation of the Application

The automated data reporting application developed in this study has been evaluated based on key performance indicators such as processing speed, accuracy, scalability, and overall impact on operational efficiency. The application integrates multiple data sources—Salesforce, Google Analytics, Azure Table Storage, and PostgreSQL—into a unified interface that delivers real-time reporting through a user-friendly React-based dashboard.

From a technical standpoint, the use of Node.js with Express.js facilitates seamless API interactions, while PostgreSQL enhances structured data handling. Docker containerization, deployed via Azure Container Apps, ensures high availability and ease of scaling. Integration with cloud-based services enables the system to maintain real-time data accessibility and consistent performance under varying loads.

The backend implements a CORS policy to prevent unauthorized access from the deployed frontend domain. As this browser level protection not a real authentication mechanism, a server-side authorization check has been implemented in this project. This authorization ensures that only authorized frontend that is configured with JWT, can communicate with the backend. It can communicate from even out of the browser scope.

From Osuria's perspective, they don't need to prepare manual report anymore that saves 30 minutes per customers equivalent to 56 hours per month for the whole customer base. This is a significant improvement as the customer can access the report in real-time that reduces operational overhead.

However, the application is not without its limitations. Occasional delays due to API rate limits or synchronization lags were noted, indicating the need for further optimization. Additionally, while the current design supports Salesforce, Google Analytics, Azure Storage, and PostgreSQL, extending compatibility to other platforms would require additional development.

Overall, the application demonstrates a substantial advancement in data management practices, replacing a time-intensive manual process with a scalable, automated, and efficient reporting solution tailored to the business needs of Osuria.

13 Conclusion and Future Work

The study has shown the advantages of using an automated data reporting application tool to overcome the challenges associated with manual data processing. The developed application effectively streamlines data reporting processes, making them more efficient, accurate and scalable by leveraging modern web technologies, cloud computing, and API integrations. The transition from manual reporting to an automated system has lessened workload, reduced errors and given business insights in real-time.

However, certain areas flag further improvement for the application. Future development efforts should focus on optimizing API performance to handle large data volumes effectively to support a broader range of data sources. Additionally, implementing advanced security design, such as role-based accessibility and multi-factor authentication, could further enhance data protection.

Overall, the automated data reporting application is a great leap from traditional ways of doing things with greater scalability, efficiency, and reliability in transforming data into actionable insights. As mentioned, the presented implementation tackles critical challenges, but through continuous upgrades and optimizations, it will further strengthen its impact and become a valuable tool for the commissioning organization.

Agarwal, G., 2021. Modern DevOps Practices: implement and secure DevOps in the public cloud with cutting edge tools, tips, tricks, and techniques. Packt Publishing. Birmingham-Mumbai.

Alteryx (2023) Automated Reporting: Benefits and Features, Alteryx. Available at: https://www.alteryx.com/blog/automated-reporting (Accessed: 13 February 2025).

Alves, R. (2023). Get Started with the PERN Stack: An Introduction and Implementation Guide. Medium. Available at: https://medium.com/@rita_palves/get-started-with-the-pern-stack-an-introduction-and-implementation-guide-e33c55d09994 (Accessed: 12 February 2025).

Aniche, M. (2022) Effective software testing. Shelter Island, NY: Manning Publications.

Ayanda, I.O. (2023) Getting Started: Docker Container, Docker Image & Dockerfile. DEV Community. Available at: https://dev.to/oayanda/getting-started-docker-container-docker-image-dockerfile-2oj9 (Accessed: 12 February 2025).

Bierig, R., Brown, S., Galván, E. and Timoney, J. (2022) Essentials of software testing. Cambridge: Cambridge University Press.

Brown, E. (2014) Web Development with Node and Express: Leveraging the JavaScript Stack. Sebastopol, CA: O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/web-development-with/9781491902288/?ar= (Accessed: 15 February 2025).

Brown, E., 2014. Web Development with Node and Express: Leveraging the JavaScript Stack. Sebastopol, CA: O'Reilly Media, Inc.

Chao, L., 2013. Cloud Database Development and Management. Auerbach Publications. Available at: https://learning.oreilly.com/library/view/cloud-database-development/ [Accessed 19 Feb. 2025].

CloverDX (2023) 5 Benefits of Implementing an Automated Reporting Process in Your Organization, CloverDX. Available at: https://www.cloverdx.com/blog/5-benefits-of-implementing-an-automated-reporting-process-in-your-organization (Accessed: 13 February 2025).

Domo (2023) 7 Benefits of BI-Driven Automated Reporting for Your Business, Domo. Available at: https://www.domo.com/learn/article/7-benefits-of-bi-driven-automated-reporting-for-your-business (Accessed: 13 February 2025).

Emadamerho-Atori, N. (2024) The Good and the Bad of Express.js Web Framework. AltexSoft. Available at: https://www.altexsoft.com/blog/expressjs-pros-and-cons/ (Accessed: 24 March 2025).

Google Analytics (2023) 'Analytics Tools & Solutions for Your Business'. Available at: https://marketingplatform.google.com/about/analytics/ (Accessed: 13 February 2025).

Haakman, W. and Hooper, R. (2023) Azure Containers Explained. 1st edn. Birmingham: Packt Publishing. Available at: https://learning.oreilly.com/library/view/azure-containers-explained/9781803231051/?ar= (Accessed: 04 January 2025).

Herron, D. (2016). Node.js Web Development. Packt Publishing. 3rd ed. Birmingham-Mumbai.

Hinkula, J. (2023) Full Stack Development with Spring Boot 3 and React: Build modern web applications using the power of Java, React, and TypeScript. 4th edn. Birmingham: Packt Publishing.

Ifrah, S. (2023) Getting Started with Containers in Azure: Deploy Secure Cloud Applications Using Terraform. 2nd edn. New York: Apress. Available at: https://learning.oreilly.com/library/view/getting-started-with/9781484299722/?ar= (Accessed: 15 February 2025).

Imperor Brains, 2024. Advantages and disadvantages of Node.js: Reasons to choose or avoid Node.js. Medium. Available at: https://medium.com/@emperorbrains/advantages-and-disadvantages-of-node-js-reasons-to-choose-or-avoid-node-js-ed93cc895b48 [Accessed 20 Mar. 2025].

Jog, T.M. (2016). Learning Modular Java Programming. 1st ed. Packt Publishing. Birmingham-Mumbai.

Jones, M. B. (2015). JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. IETF RFC 7523. https://datatracker.ietf.org/doc/html/rfc7523

Katzer, J., 2020. Learning Serverless: Design, Develop, and Deploy with Confidence. O'Reilly Media, Inc. Available at: https://learning.oreilly.com/library/view/learning-serverless/9781492057000/ (Accessed 19 Feb. 2025).

Loubser, N., 2021. Software Engineering for Absolute Beginners. London: Apress Media LLC.

Martin, K. 2024.
The complete developer: master the full stack with TypeSript, React, Next.js, MongoDB, and Docker. No starch press. San Francisco

Microsoft (n.d.) Azure PostgreSQL Flexible Server documentation. Available at: https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/ (Accessed: 21 February 2025).

Microsoft Azure (2023) 'What is Azure?'. Available at: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure (Accessed: 13 February 2025).

Mili, A. and Tchier, F. (2015) Software testing: concepts and operations. Hoboken, NJ: John Wiley & Sons.

OWASP Foundation. (2024). OWASP Cheat Sheet Series: Authentication Cheat Sheet. Retrieved from https://cheatsheetseries.owasp.org/

Pawar, P. (2025) PERN Stack vs MERN Stack Differences. Aalpha Information Systems. Available at: https://www.aalpha.net/blog/pern-stack-vs-mern-stack-differences/ [Accessed 23 Feb. 2025].

PMC (2013) Automation of Data Collection: Improving Data Quality and Speed, National Center for Biotechnology Information. Available at: https://pmc.ncbi.nlm.nih.gov/articles/PMC3644815/ (Accessed: 13 February 2025).

Powers, S., 2012. Learning Node. Sebastopol, CA: O'Reilly Media, Inc.

Purewal, S., 2014. Learning Web App Development. 1st ed. Sebastopol, CA: O'Reilly Media, Inc.

Quadri, M.H.S., 2024. Analysing Linus Torvald’s critique of Docker. OpenSourceForU. Available at: https://opensourceforu.com/2024/12/analysing-linus-torvalds-critique-of-docker/ [Accessed 20 Mar. 2025].

Salesforce (2023a) AI for CRM. Available at: https://a.sfdcstatic.com/content/dam/www/ocms/assets/pdf/misc/ai-for-crm.pdf (Accessed: 13 February 2025).

Salesforce (2023b) The Future of AI in CRM: Trusted, Limitless, Personalized. Available at: https://www.salesforce.com/content/dam/web/en_au/www/documents/pdf/salesforce-future-ai-tlp.pdf (Accessed: 13 February 2025).

Satheesh, M., D'mello, B.J. and Krol, J. (2017). Web Development with MongoDB and NodeJS. 2nd ed. Packt Publishing. Birmingham-Mumbai.

Savil, J., 2020. Microsoft Azure Infrastructure Services for Architects Designing Cloud Solutions. Indiapolis: John Wiley & Sons, Inc.

Scott, A.D., 2020. JavaScript Everywhere: Building Cross-Platform Applications with GraphQL, React, React Native, and Electron. Sebastopol, CA: O'Reilly Media, Inc.

Sengupta, D., Singhal, M. and Corvalen, D. (2016). Getting Started with React. 1st ed. Packt Publishing. Birmingham-Mumbai

Sharma, N., 2024. How to dodge these 7 common mistakes in ReactJS Development. Kellton. Available at: https://kellton.com/kellton-tech-blog/7-reactjs-development-mistakes-to-avoid [Accessed 20 Mar. 2025].

Shklar, L. & Rosen, R., 2009. Web Application Architecture: Principles, Protocols and Practices. 2nd ed. Hoboken, NJ: Wiley. Available at: https://learning.oreilly.com/library/view/web-application-architecture/9780470518601/ [Accessed 19 Jan. 2025].

Stuttard, D. and Pinto, M. (2011) The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. 2nd edn. Indianapolis: Wiley Publishing.

Vadlamani, V. (2024). PostgreSQL Skills Development on Cloud: A Practical Guide to Database Management with AWS and Azure. Apress. Available at: https://www.google.fi/books/edition/PostgreSQL_Skills_Development_on_Cloud/qC82EQAAQBAJ?hl=en&gbpv=1&dq=azure+postgresql+flexible+server&pg=PA237&printsec=frontcover (Accessed: 12 February 2025).

Wittig, A. & Wittig, M., 2015. Amazon Web Services in Action. Shelter Island, NY: Manning Publications. Available at: https://learning.oreilly.com/library/view/amazon-web-services/9781617292880/ [Accessed 19 Feb. 2025].