Tamanji Che

Optimisation of the OTCM Application through Cloud Database Migration and Refactoring

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

1 May 2024

Abstract

Author:	Tamanji Che
Title:	Optimisation of the OTCM Application through Cloud Database Migration and Refactoring
Number of Pages:	65 pages + 2 appendices
Date:	1 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Jouni Miikki, Software Solutions Lead Toni Spännäri, Senior Lecturer

This thesis aimed to enhance the performance of the Operational Technology Condition Monitoring (OTCM) application through strategic cloud database migration and refactoring. It was carried out for the case company, Siemens Oy. The OTCM application, integral for monitoring industrial devices across various sites, experienced significant performance problems, particularly unresponsiveness in the user interface and latency issues with an external API. These limitations necessitated migrating from the existing external database to a more scalable internally managed PostgreSQL cloud database.

The methodology encompassed a comprehensive analysis of the existing system, identification of performance bottlenecks through performance testing tools such as Locust and Chrome DevTools, and subsequent database migration and source code refactoring. The new system architecture aimed to enhance data processing speeds, reduce latency, and improve overall application responsiveness and user experience.

The results indicated a significant improvement in performance metrics such as response time and system throughput. The successful migration and refactoring not only optimised the OTCM application but also enhanced the maintainability and scalability of the application, ensuring the reliable monitoring of operational technology devices.

This study demonstrates the effectiveness of cloud database migration and application refactoring in resolving performance issues, laying the groundwork for future improvements in similar industrial applications.

Keywords: Data-Migration, refactoring, optimisation, performance

Contents

List of Abbreviations

1 Introduction 1

2 Background on OTCM Current State Analysis 3

2.1 Software Development Tools 3
2.1.1 TypeScript 3
2.1.2 NodeJS 4
2.1.3 Siemens Insights Hub 4
2.2 Software Performance 5
2.2.1 Response Time and Latency 5
2.2.2 Throughput 6
2.2.3 Scalability 8
2.2.4 Resource Utilisation 9
2.2.5 Availability 10
2.3 Software Performance Testing 11
2.3.1 Performance Testing Types 12
2.3.2 Performance Testing Methodology 13
2.4 Performance Testing Tools 15
2.4.1 Locust 15
2.4.2 DevTools 17

3 Current State Analysis of the OTCM 21

3.1 Description of OTCM 21
3.2 The Identified Problem 23
3.3 Performance Evaluation of OTCM 23
3.3.1 Test Plan for Identifying Bottlenecks in OTCM 24
3.3.2 Initial Performance Assessment with DevTools 25
3.3.3 Load Testing with Locust 29
3.3.4 OTCM Performance Goals 33
3.3.5 Recommended Optimisation Strategies 34

4 Theory on Database Migration and Refactoring 35

4.1 Database and Database Migration 35
4.1.1 Designing a Database 38
4.1.2 Database Optimisation 40
4.1.3 Data Integrity and Database Transaction 42
4.1.4 Data and Schema Migration 43
4.1.5 Migration with Knex 45
4.2 Refactoring 46

5 Optimisation Target / Proposed Solution 47

6 Implementation 50

6.1 Migration of OTCM Database Schema 50
6.2 Database Service Implementation 51
6.3 Data Migration 53
6.4 Source Code Refactoring 53
6.5 Performance Tuning 54
6.6 Performance Testing of the Optimised OTCM 55
6.6.1 OTCM Database API Stress Testing 56
6.6.2 Web Performance Assessment 59
6.7 Summary of OTCM Performance Improvements 61

7 Conclusion 62

References 63

Appendices
Appendix 1: Locust Load Testing Scripts for OTCM
Appendix 2: Data Migration Scripts for OTCM Database

List of Abbreviations

API: Application programming interface. A set of rules that allows different software entities to communicate with each other.

CLI: Command line interface. A text-based interface for operating software and devices.

CPU: Central processing unit. The primary component of a computer that performs most of the processing inside a computer.

DBMS: Database management system. Software for maintaining, querying, and updating data and metadata in a database.

ER: Entity relationship. A model defining the relationships among entities in a database.

FPS: Frames per second. In web browsers, FBS refers to the rate at which web content is rendered to the display or the number of frames that can be repainted in a second.

HTTP: Hypertext transfer protocol. An application protocol used for transmitting hypermedia documents, such as HTML.

IoT: Internet of things. A network of interconnected devices that communicate over the internet.

I/O: Input/output. Refers to the communication between an information processing system (like a computer) and the outside world, possibly a human or another information processing system.

OLTP: Online transaction processing. A benchmark that refers to the computational operations involved in processing transactional requests from users in a software system.

OT: Operational technology. Hardware and software that detects or causes a change through the direct monitoring and control of physical devices, processes, and events.

OTCM: Operational technology condition monitoring. An internal application for the case company for monitoring the condition of equipment and systems.

PG: PostgreSQL. An open-source relational database management system emphasising extensibility and SQL compliance.

RDBMS: Relational database management system. Software for maintaining, querying, and updating data and metadata in a relational database.

RPS: Requests per second. A measurement of the throughput of a system, indicating the number of requests a system can handle per second.

SQL: Structured query language. A domain-specific language used in programming and design for managing data held in a relational database management system.

TBT: Total blocking time. The total amount of time that a page is blocked from responding to user input.

TCP: Transaction processing performance council. A non-profit organisation that defines transaction processing and database benchmarks.

TTFB: Time to first byte. The time it takes for a browser to receive the first byte of data from the server.

UI: User interface. It defines how a user and a computer system interact.

1 Introduction

This thesis evaluates cloud database migration and refactoring to optimise the existing Operational Technology Condition Monitoring (OTCM) application for the case company, Siemens Oy. The OTCM application is part of an ecosystem for monitoring the health of Operational Technology (OT) devices installed at different sites. The unresponsiveness of parts of the User Interface (UI), and the increased response time of the Event Management Application Programming Interface (API) of Insights Hub and API - throttling by Insights Hub, resulted in a reduced user experience. This necessitated the migration of the data of the OTCM application from the Event Management database (of Insights Hub) to the PostgreSQL (PG) cloud database for improved performance.

Database migration refers to the transfer of data from one database to another. Migrations may result in performance and scalability improvements. Conversely, refactoring entails modifying source code to enhance maintainability, readability, and performance.

The main objective of this thesis is to improve the performance, maintainability, and consequently the user experience of the OTCM application. Performance testing was conducted on the OTCM both before and after the optimisation of the application, and the results were compared to verify improvements.

This thesis consists of seven sections: The Introduction is followed by the Background on OTCM Current State Analysis section which explores some theories on the tools and technologies utilised to analyse the current state of the OTCM application. The third section, Current State Analysis of OTCM, provides an overview of the state of the OTCM and investigates the performance problems identified on the OTCM. The fourth section, Theory on Database Migration and Refactoring, explores the theory that supports the proposed optimisation solution for the OTCM.

This is followed by the fifth section, the Optimisation Target which discusses the proposed solution. The sixth section, the Implementation, outlines the execution of the proposed solution and the tests carried out. The last section concludes the thesis with an interpretation of the results and recommendations for future improvements.

2 Background on OTCM Current State Analysis

This section explores concepts and methods such as software performance and performance testing, providing an overview of the tools utilised in the development of the OTCM application, and the analysis of the current state of the application (which is presented in the following section, Current State Analysis of the OTCM).

2.1 Software Development Tools

The OTCM application is a complex software developed using multiple modern tools, including TypeScript, NodeJS, and Siemens Insight Hub. This subsection provides an overview of these technologies.

2.1.1 TypeScript

TypeScript is an open-source language, a statically typed superset of JavaScript. Its incorporation of static type-checking allows for enhanced code quality and early identification of potential type-related errors during the compile-time phase. TypeScript offers numerous advantages, including but not limited to, enhanced code readability, maintainability, and robustness. TypeScript consists of both the language and the compiler. The language augments JavaScript with static typing and object-oriented programming features, while the compiler converts TypeScript into native JavaScript codes, facilitating the development of error-free codes. TypeScript promotes better software design, making the codebase more understandable, easier to refactor, and less prone to bugs. [1,2].

Beyond type safety, TypeScript introduces interfaces, generics, and advanced type annotations, allowing developers to harness Object-Oriented Programming principles. In addition, TypeScript offers interoperability with JavaScript, allowing the incremental adoption of TypeScript by projects with existing JavaScript codebases. [3]. Thus, TypeScript is vital for developing and maintaining a complex system such as the OTCM application.

The mechanism of converting a source code from one language to another is called transpilation. In the context of TypeScript, it refers to the conversion of codes from TypeScript to JavaScript. As a development tool, all TypeScript codes are transpiled to JavaScript before deployment to production. TypeScript does not exist at runtime. The transpiled codes are then executed by a JavaScript engine in an environment such as a web browser or NodeJS. [2].

2.1.2 NodeJS

NodeJS was created to run JavaScript code on the server side as an open-source, cross-platform runtime environment. It was developed with an emphasis on enhancing the scalability of networked applications, it diverges from traditional application platforms by utilising a single-threaded, event-driven architecture, instead of multi-threaded processing. This unique approach, which avoids the complexity of threads, aims to enhance throughput and reduce latency, offering a simpler programming model. At its core, Node.js operates on the V8 JavaScript engine, originating from the Chrome web browser. However, there are efforts to diversify this reliance, such as an implementation of the ChakraCore JavaScript engine of Microsoft. While Node.js is not an application server, it provides the foundational runtime for executing JavaScript, augmented by non-blocking I/O operations and a robust ecosystem for general-purpose programming. [4,5].

2.1.3 Siemens Insights Hub

Insights Hub is a cloud-powered Internet of Things (IoT) operating system designed by Siemens to optimise industry, infrastructure, and city operations by harnessing the potential of interconnected devices. Insights Hub connects products, plants, systems, and machines through IoT, generating actionable insights, thus promoting innovation, efficiency, and increased productivity. [6].

Insights Hub offers extensive connectivity options, integrating various devices and systems. The effective data analytics capabilities of Insights Hub translate large volumes of data into beneficial business insights. Additionally, the Platform-as-a-Service environment of Insights Hub allows developers to construct, deploy, and operate cloud-based applications seamlessly. [6,7].

2.2 Software Performance

This subsection provides an overview of software performance and testing, establishing the theoretical foundation for analysing and optimising the OTCM application, as discussed in subsequent sections.

Software performance is defined as the efficiency and effectiveness of a software application or system within specified constraints such as processing speed, resource utilisation, and responsiveness. It is characterised by metrics such as response time, latency, throughput, and scalability. Furthermore, performance is influenced by the design and architecture of the software, the underlying hardware, and network considerations. [8, 9, 10, 11].

2.2.1 Response Time and Latency

Response time is a measure of the time interval required by a system to respond to a user-initiated request or action. Response time is often influenced by various forms of latency, such as network, database, and processing latencies. Network latency refers to the data-transfer time over a network. In contrast, database latency is the time it takes to retrieve data from a database, and processing latency measures the time for processing the request within the software. [10, 13].

A minimal response time correlates positively with the responsiveness of an application, resulting in an adequate user experience. Delays ranging from two to four seconds can impede concentration-intensive operations. However, a delay in that same range is considered acceptable in situations such as after entering the payment details during product purchase, but the same response time is inadequate when users are comparing product features. Furthermore, the response time for tasks requiring memory retention across multiple responses is expected to be within two seconds. [10].

The success or unsuccessfulness of the performance of a system is determined by the variance from an acceptable baseline of the response time at the targeted throughput. For instance, assuming a baseline of seventy-five seconds, a five hundred seconds response time with a load of a thousand users is considered unacceptable. Conversely, a comparatively small increase to two-hundred and fifty seconds of the response time is considered adequate for maintaining end-user productivity. While an increase in response time is expected as the load on a system increases, it should not be directly proportional to the additional load. [10].

2.2.2 Throughput

Throughput refers to the number of transactions, requests or operations successfully processed by a system within a specific period. The specific operation depends on the system being tested. As a result, operations are clearly defined to ensure the significance of the throughput value. For instance, industry benchmarks such as the TPC-C benchmark defined by the Transaction Processing Performance Council (TPC) for Online Transaction Processing (OLTP) systems, provide a standardised method for measuring and reporting throughput. TPC-C is an OLTP benchmark that evaluates the number of orders processed per minute. The results of this benchmark are expressed as transactions-per-minute-C (tpmC). [13, 14].

OLTP refers to the computational operations involved in processing transactional requests from users in a software system. An OLTP workload depicts the different tasks performed by the users in question. This workload is usually defined by the types of user activities and the corresponding number of users engaged in each activity. [9].

Regarding the performance testing and tuning of the OTCM application, throughput is represented as the number of successful Hypertext Transfer Protocol (HTTP) requests processed per second. The curve below illustrates throughput as a function of the load on a system.

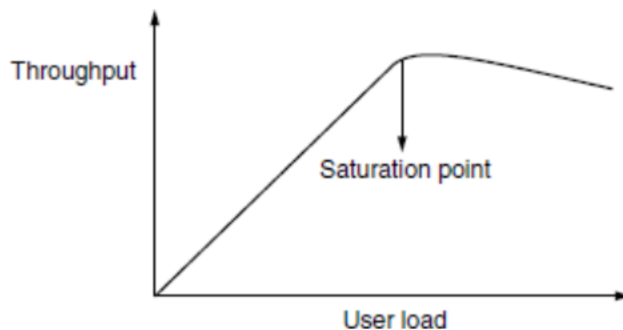

Figure 1. Throughput as a Function of Load. Copied from Desikan and Ramesh (2007) [15].

As depicted by Figure 1, throughput increases linearly with increasing user load on the system until a maximum value is attained at the saturation point. Beyond the saturation point, the throughput decreases with increasing user load. This part of the curve indicates decreasing performance, implying that the resources of the system such as memory and the Central Processing Unit (CPU) are being fully utilised or that limitations exist in the system. As a result, the response time increases and is perceived as inadequate by users. The linear part of the curve is typical of an efficient system, indicating the potential of the system to operate adequately at increased loads. [13, 15].

The optimal throughput of the system is designated by the saturation point, signifying the maximum capacity of the system [13, 15]. According to Molyneux (2014), the throughput of a system in combination with the response time is essential in assessing the system and ensuring a good user experience [10]. Additionally, throughput is correlated to the scalability of a system [13].

2.2.3 Scalability

Another significant performance metric is scalability. A system is considered scalable when the performance of the system remains relatively unaffected by an increase in the number of users or an equivalent increase in the load of the system [13].

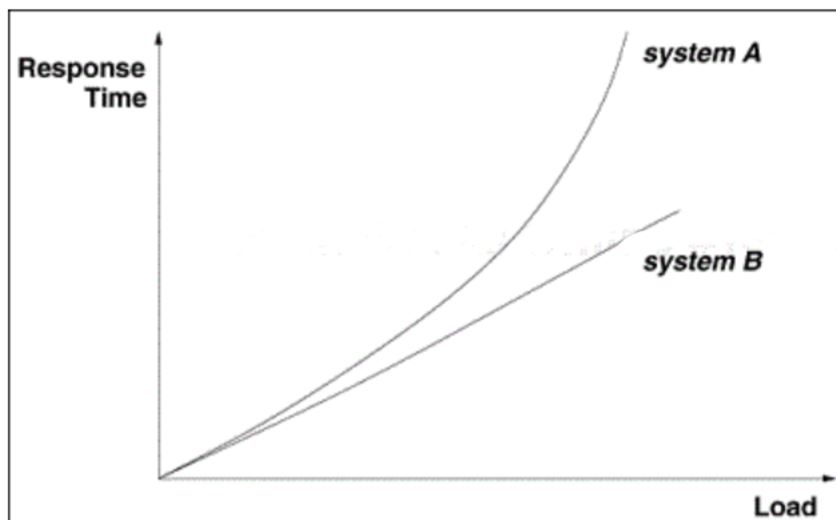

Figure 2. Comparing Two Systems for Scalability. Copied from Almeida, Dowdy and MenascÃƒÆ’Ã‚Â (2004) [13].

As illustrated in Figure 2, the correlation between the response time and the load for both System A and System B demonstrates that System B is scalable and relatively efficient compared to System A which is unscalable. The response time of System A increases exponentially with increased load, indicating the inefficiency and non-scalability of the system. [13].

In contrast, the linear curve of System B indicates a proportional and gradual growth of the response time with increased load on the system. Consequently, System B is scalable and more efficient than System A. [13].

2.2.4 Resource Utilisation

The performance of a system is also affected by the resources available to it. The measure of the efficiency of a system in utilising available resources such as CPU, memory, storage, and network bandwidth is defined as resource utilisation. CPU specifications such as architecture, clock speed, number of cores, cache size, and memory bus speed, are vital metrics for determining the ability of a system to process tasks efficiently. Memory is another critical factor; the primary consideration is the total memory available to the system. Thus, the amount of data that is processed and stored temporarily during operations is directly affected by the available memory. Furthermore, the cache at the storage level is instrumental in enhancing Input/Output (I/O) performance by reducing the time required to access data. Storage is further characterised by factors such as the total number of I/O controllers, the array of ports on each controller, and the total count of physical disks. [9, 10].

Bandwidth is generally sufficient within modern data centres. However, as transmissions approach the end user, network problems such as congestion and interference may occur. Consequently, performance may significantly deteriorate, particularly for communications over the Internet. Network utilisation is typically characterised by data throughput, data error rate, and data volume metrics. Data volume refers to the quantity of data that is transferred over a network. Large data volumes can be problematic over low-bandwidth connections, resulting in decreased performance due to bandwidth restrictions and latency. In contrast, data throughput refers to the actual rate at which data is successfully transferred over a network. Monitoring data throughput is essential in achieving performance targets such as page requests per second. [10].
A significant decrease in data throughput is often an indication of capacity limitations, which can impair connectivity and degrade the user experience.

Additionally, data errors that necessitate retransmission may consume more bandwidth and time, further reducing effective throughput. Therefore, determining if the errors originated from the application or the network infrastructure is imperative. [10].

2.2.5 Availability

An efficient system or application is further characterised by availability, ensuring user access. Availability is a measure of the percentage of time that a system is operational and accessible for handling requests. It is typically assessed by external monitoring agents that perform periodic verifications of the responsiveness of the system. For instance, the connectivity of an online bookstore may be tested by pinging the services of the online store. Additionally, the comprehensive availability of the services of the online store may be evaluated by software agents across different locations. This evaluation is achieved by requesting different services from the bookstore, such as a book search. While constant availability is imperative for user satisfaction, scheduled maintenance periods are a necessary and planned exception. [10, 13].

Nevertheless, a successful connection to a web server is not necessarily a confirmation of the operational status of the entire application. While the browser may establish a connection to the web server, the homepage of the application may be inaccessible. In addition, an application may operate seamlessly with a minimal load but perform inadequately with increased loads, indicating a possible problem in scalability and not limitations in the functionality of the application. [10].

As affirmed by Molyneux (2014), an efficient application enables a seamless and responsive user experience, allowing the completion of tasks by users without delays or frustrations [10]. Wagner (2016) further affirmed that optimally performing websites enhance user experience and engagement by promptly rendering content [12].

In contrast, user interest is observed to be significantly reduced by prolonged page loading times, resulting in a decrease in website traffic and consequently a reduction in revenue. In e-commerce, the expectations of users regarding the load time of websites are elevated. Fifty per cent of users are reported to have expected a page load time lower than two seconds, and forty per cent of users have been disengaged from websites by a loading time that exceeds three seconds. Additionally, it has been established that a one-second delay in page response resulted in a seven per cent reduction in user activity on the platform. [12]. Consequently, evaluating, and optimising software performance is vital for ensuring user satisfaction and achieving adequate operational objectives.

2.3 Software Performance Testing

Performance testing is defined as the evaluation of the responsiveness, reliability, throughput, interoperability, and scalability of a system or application within specified constraints. The objective of performance testing is to assess the efficiency and effectiveness of a computer, network infrastructure, or software application. Usually, tests are automated with testing tools, facilitating the repeatability of simulating normal and unusual load conditions. Additionally, performance testing is employed to identify bottlenecks and errors in a system. [11].

Non-performant applications are unreliable, resulting in the expenditure of both financial and time resources. Performance testing is imperative in ensuring that the designated functions of applications are fulfilled reliably and efficiently, thereby conferring the expected user experience. Common types of performance testing that are employed include load, stress, soak, and reliability tests. [10, 11, 13].

2.3.1 Performance Testing Types

Load testing is the process of evaluating the performance of a system by subjecting it to increased loads and measuring the response and capacity of the system under expected conditions. This is achieved with tools such as Locust, which supports the simulation of millions of users, thus providing load conditions that represent the actual use case. [11, 16]. Locust is discussed in the subsequent subsection, Performance Testing Tools.

Contrary to load testing, stress testing applies loads that exceed the specified operational capacity or resource limits of the system. During testing, the system is deliberately subjected to challenging conditions such as insufficient memory, inadequate hardware or an unusually large number of transactions or users. Stress testing is performed to observe the responsiveness of a system and identify the thresholds at which the efficiency of a system is unacceptable under unusual conditions. [10, 15].

According to Desikan and Ramesh (2007), the performance of a system is expected to degrade gradually with increasing load. However, the unexpected and complete unresponsiveness of a system during stress testing is unacceptable. [15]. Hence the system or application is expected to maintain a minimal level of functionality and not to be completely unavailable during stress testing.

However, Molyneaux (2014) acknowledges the potential for the total unavailability of a system as a critical outcome of stress testing. The unavailability is a possible indication that the maximum capacity of the system has been exceeded. This is essential in understanding the capacity and performance of the system. [10]. Thus, stress testing may result in varying levels of system degradation, from a gradual decrease in efficiency to complete unavailability or unresponsiveness. The result is dependent on the resilience and design of the system. The primary outcome is identifying the breaking point of the system.

Soak testing involves subjecting a system to sustained loads for an extended period. The extended duration is essential in revealing problems such as progressive memory leaks and resource limitations that may only occur due to a system sustaining a significant number of operations or activities. Adequate infrastructure monitoring is imperative for successful soak testing, as the correlation of the data from both the added load and the infrastructural resources at the point of system failure or unresponsiveness is critical in ensuring an accurate assessment of the test results. [10]. Thus, soak testing is designed to ensure that the expected operational performance of a system is sustained for extended periods.

2.3.2 Performance Testing Methodology

The performance testing process is typically comprised of the following main steps: scoping, planning, and designing the test, execution and monitoring of the test, analysis of the test results, and tuning or optimisation of the software application. During the first phase, the criteria for acceptable performance and the test environment are established, guiding the planning, and design of the tests. This critical step involves a comprehensive knowledge of the user expectations, business context, and technical limitations of the application. Key Performance Indicators (KPIs) such as response time, throughput, and resource utilisation are defined at this stage. For instance, defining acceptable response times based on user tolerance levels and aligning throughput with business requirements. [9, 10, 11].

In addition, the test environment is designed to accurately simulate the production environment, ensuring that the test results adequately represent the application usage. This involves configuring the hardware, software, network, and other infrastructure elements to match the actual environment where the application is run. Furthermore, the tests are designed to simulate the actual user interactions of the application, ensuring that all critical features are thoroughly tested. [11].

In the second phase, the designed test cases are scripted and executed with the preferred testing tool such as Locust within the pre-configured test environment. It is recommended to include resource monitoring in test configurations to enable a proper analysis of the obtained results. During test execution, it is essential to monitor for errors as large error rates may signify problems in the test script, the application, or the resources of the system. [11].

The Repeatability of executed tests is crucial in ensuring the significance and adequacy of test results. Conducting multiple iterations of the test is necessary to confirm that the discrepancies between results from different repetitions are minimal or non-existent. A significant variation in results across repetitions may imply unreliable tests. According to Liu (2009), assessing the repeatability of tests under identical conditions involves conducting a series of initial tests, verifying the reliability of the hardware and system configurations, and ensuring consistency in the results. While minor fluctuations in results (within a five per cent margin) are generally acceptable, larger inconsistencies (exceeding a ten per cent threshold) require investigation to identify and resolve the problem. [9].

Thirdly, the results from the conducted tests are analysed to identify the problems that are affecting the performance of the application. The identified bottlenecks may result from the database layer, the application, or the system infrastructure such as CPU, memory, and network. The sources of the problems are vital in deciding on the necessary implementations or strategies for optimising the application and tuning system resources. [9, 11].

Finally, the optimisation strategies are applied to achieve the targets defined in the first phase of the testing process. Bottlenecks originating from the system infrastructure are resolved by actions such as increasing memory allocation, optimising CPU usage and adjusting network configurations. Database-level bottlenecks are resolved by applying performance strategies such as query optimisations, indexing, and adjusting database configurations. [11].

On the application level, performance optimisation may include, refactoring the source code of the application, improving the efficiency of memory consumption by the application and minimising database interactions. It is imperative that following each adjustment, tests are re-conducted to evaluate and compare the effects to previous results. Implementing changes sequentially allows for a clear assessment of the impact of each modification. This iterative process of testing and adjusting is continued until the performance objectives are attained. [11].

2.4 Performance Testing Tools

This subsection provides an overview of the tools utilised in assessing the performance of the OTCM application.

2.4.1 Locust

Locust is an open-source framework for load testing that is written in the Python programming language [16]. The following capabilities characterise it:

- Real-time monitoring is supported by Locust. Statistics regarding the number of requests per second, response times, and the number of failures, which are essential metrics for performance testing, are provided in real time [16].
- Locust is distributed and scalable. It is capable of handling numerous simultaneous users on a single or multiple machines [16].
- Locust enables the weighting and scheduling of tasks or operations. Different types of user actions may be weighted to accurately depict the types of requests typically encountered in an application [16].
- It provides flexibility in scripting test cases and simulating user interactions, such as performing searches [16].
- It is platform-independent, designed to operate across different systems or platforms [17]. Examples of such systems include Linux and Windows.

Load testing with Locust is achieved by first writing a test script which defines the transactions or activities of the simulated users. Secondly, the execution of the scripted test cases is then initiated through the command line interface (CLI) of Locust and parameters such as the number of concurrent users and hatch rate are specified. The hatch rate refers to the frequency of adding users to the application being tested. During the test execution, real-time statistics regarding the performance of the application are monitored through the dynamic web UI or the CLI. Finally, the result of the completed test is reported as charts and statistical data which is then analysed for optimisation. [16]. A sample of the result of a Locust load test is presented in Figure 3 below.

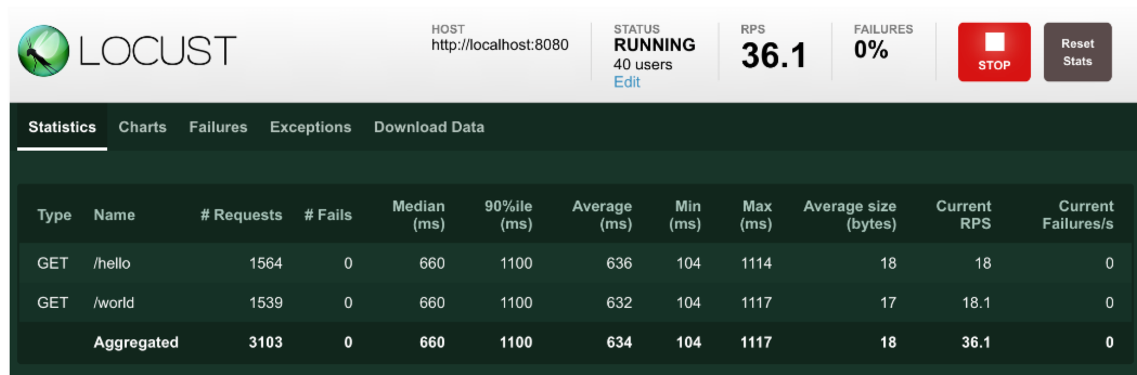

Figure 3. Locust Load Test Result. Copied from Locust Documentation [16].

As illustrated in Figure 3, the statistical data and chats of the results of a Locust test are presented on the web UI. The highlighted Statistics tab presents the statistics for the different types of requests, the total number of each type of request, the number of unsuccessful requests (Fails) and the number of Requests Per Second (RPS). Graphical presentations of the results are accessed through the Charts tab, and the Failures tab provides details on the errors encountered during the testing. Furthermore, the web UI contains controls such as the Stop and Reset buttons for managing the load testing. [16]. According to Siddhant and Prapulla (2020), Locust is more efficient as a testing tool compared to other open-source testing tools such as Gatling and JMeter [17]. This performance advantage is a consequence of the effective architecture and design principles of Locust [16,17]. Thus, Locust is the preferred tool for performance testing in the context of this thesis.

2.4.2 DevTools

Chrome DevTools represents a collection of development tools such as the Network and Performance panels, which are integrated into the Google Chrome browser. DevTools are instrumental in accelerating the development process, identifying bottlenecks, and improving the quality of web-based applications or platforms. The Performance panel is utilised for the analysis of both runtime and loading performances of a web page of an application. While runtime performance metrics are recorded during user interaction with the loaded page, loading performance is evaluated based on the activities that occurred during page loading. [18, 19]. Figure 4 below represents the recording of the performance of a web page with DevTools.

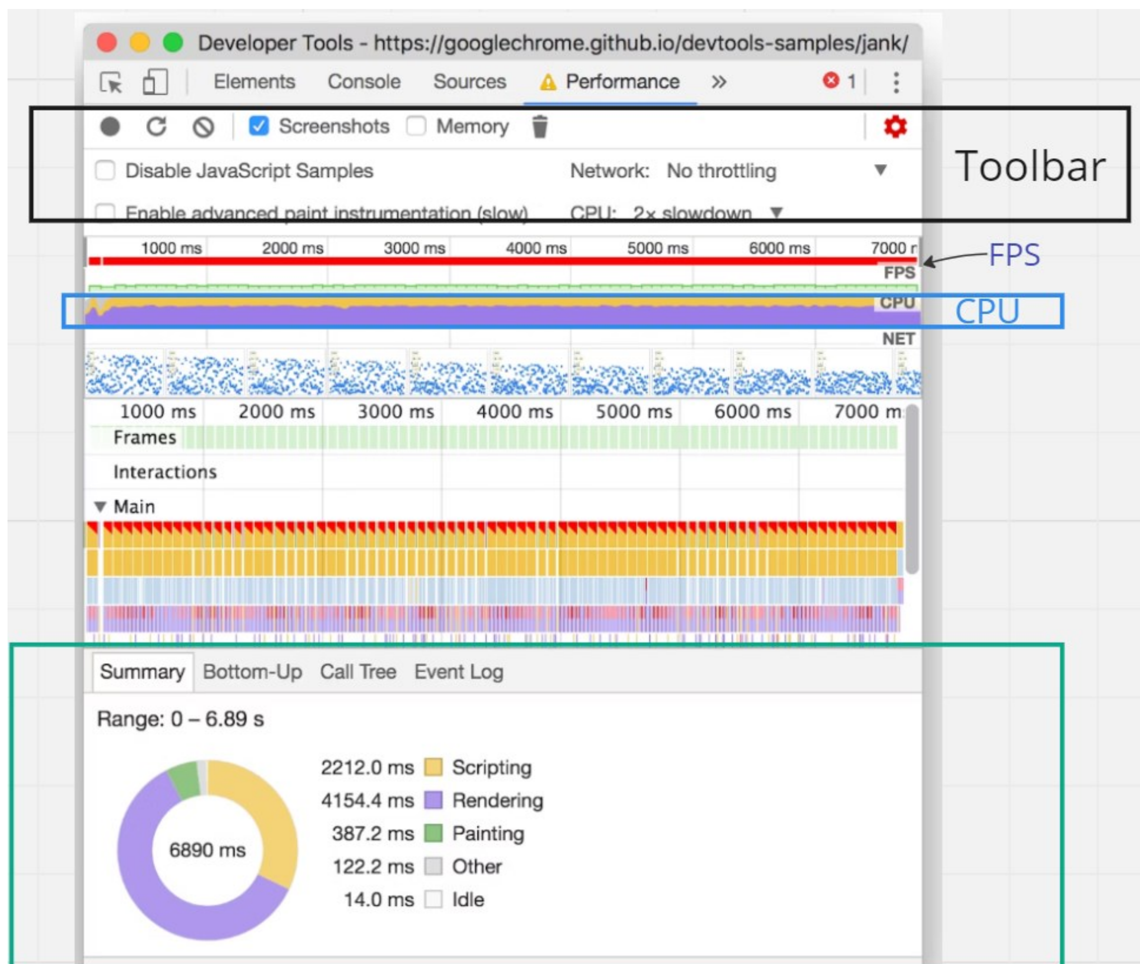

Figure 4. Record of Web page Performance. Modified from Basques (2017) [19].

As depicted in Figure 4, the different metrics and activities of the performance evaluation of a web page are demonstrated by the Performance panel of the DevTools UI. An overview of the activities of the web page that occurred during performance recording is illustrated by the timeline overview portion of the UI which is below the toolbar. The timeline overview is typically characterised by the Frames Per Second (FPS), CPU and network charts. FPS is essential in assessing the user experience of the web page. The red bar in the FPS chart indicates a significant decrease in FPS value and a corresponding decline in user experience. Generally, the magnitude of the green bars correlates positively with FPS and the resulting responsiveness of the web page. The CPU chart highlighted by the blue rectangle represents the operations executed by the CPU. The different operations are colour-coded and are presented in the Summary tab (which is emphasised by the green rectangle) at the bottom of the UI. [18, 19]. The following metrics are depicted in the Summary tab:

- Loading represents the time during which network-related or external resources such as images, stylesheets, and scripts are loaded by the browser [12, 19].
- Scripting refers to the time allocated to executing JavaScript code. Monitoring scripting time is crucial for identifying bottlenecks and inefficiencies in code execution as JavaScript is fundamental in web interactivity and functionality. [12, 18].
- Rendering signifies the time expended on critical tasks such as performing layout calculations and preparing the web page for painting [12, 19].
- Painting is a measure of the duration required to draw the layout and views of the web page on the screen [12].
- Idle time represents periods where the browser is not actively engaged in processing tasks related to the web page. Understanding this metric is beneficial in optimising resource allocation and improving the efficiency of the application. [18, 19].

Other metrics reported by the Performance panel include long tasks and Total Blocking Time (TBT). A long task requires substantial processing time, thereby obstructing the execution of other tasks, and resulting in a significant decrease in the responsiveness of a web page. Similarly, TBT quantifies the cumulative duration during which the main thread of the browser is obstructed by tasks, potentially impeding user interaction and responsiveness. [18, 19].

In contrast to the Performance panel, the Network panel is designed for investigating network requests and assessing information such as the size and timing of the different transmitted resources [12, 20]. Figure 5 below is a screenshot demonstrating the inspection of the network activity of a web page.

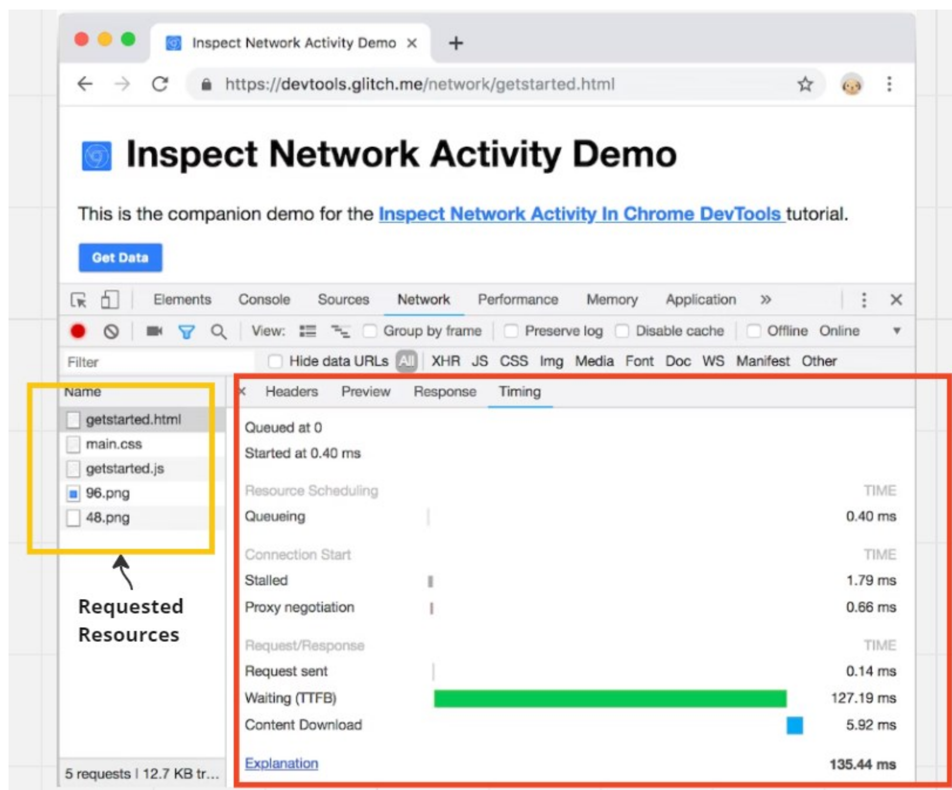

Figure 5. Timing Statistics of a Network Request. Modified from Basques and Emelianova (2015) [20].

In Figure 5 above, details of the timing information of a specific request are highlighted by the red rectangle. The green bar is a waterfall chart representing a vital metric, the Time to First Byte (TTFB). TTFB is a measure of the time interval between the initiation of an HTTP request by a browser and the reception of the response data from the web server. TTFB inversely correlates with the responsiveness and efficiency of processing requests by a server. Contrary to TTFB, page load time refers to the total duration required for the assets of a web page to be completely and successfully requested and rendered by a browser. A complete list of the resources requested for rendering the web page is accentuated by the yellow rectangle. A total of five requests were executed as indicated at the bottom of Figure 5. [12, 20].

In summary, this section provided a background for the current state analysis of the OTCM application, detailing technologies such as TypeScript, NodeJS, and Siemens Insights Hub that the application is composed of. It explored key characteristics of software performance including response time, latency, throughput, scalability, resource utilisation, and availability, which ensure the efficiency and effectiveness of an application. Furthermore, the significance of software performance testing is emphasised, highlighting the utilisation of tools such as Locust and methods such as load, stress, soak, and reliability tests in identifying and resolving potential bottlenecks. These testing methods are crucial in ensuring the operational efficiency of an application or system.

This thesis utilised Locust and DevTools to evaluate application performance and attain optimisation targets for the OTCM application. Although resource monitoring is particularly recommended for soak and stress testing (as discussed in the ‘Performance Testing Types’ subsection), it was not conducted in this study due to resource constraints and scope limitations.

3 Current State Analysis of the OTCM

This section provides an overview of the OTCM application, defining the performance problems existing in the current state of the application. Bottlenecks are identified through performance testing, source code review and analysis.

3.1 Description of OTCM

OTCM was designed by the case company, Siemens Oy, as an internal application to monitor the state and health of industrial devices installed at different customer sites, providing maintenance support for these devices. The current state of the OTCM application is presented in Figure 6 below.

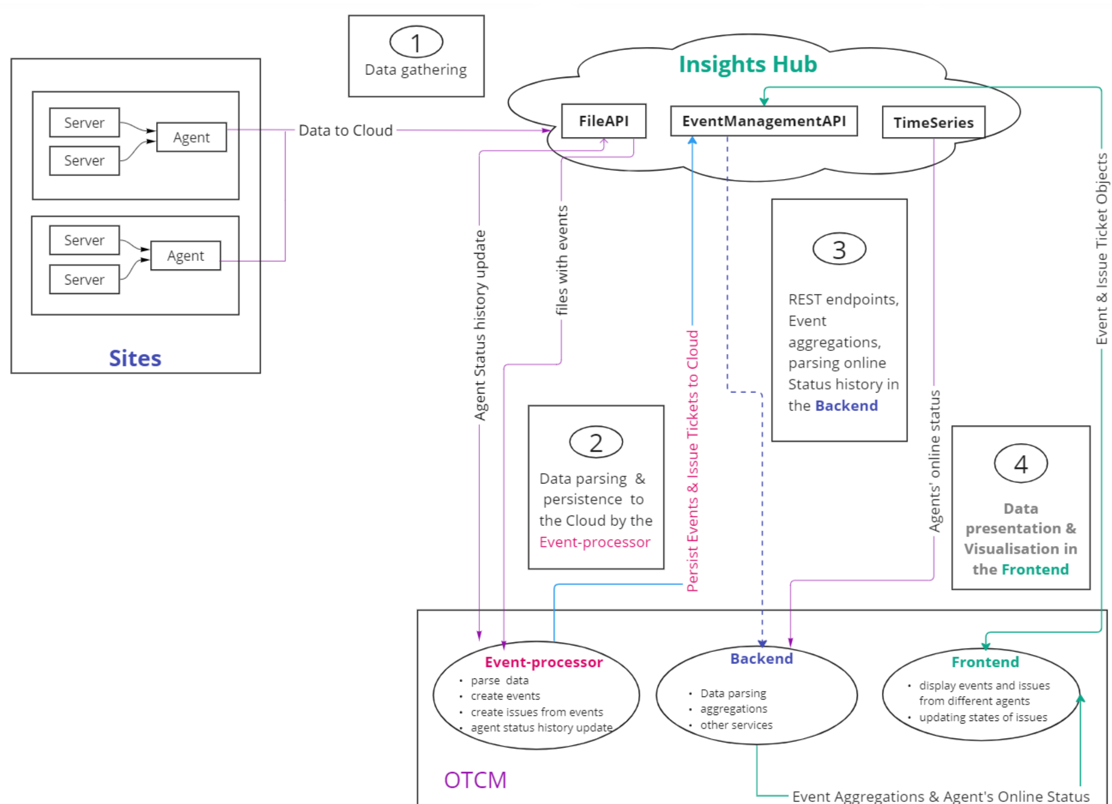

Figure 6. Current OTCM System Architecture.

As depicted in Figure 6, the system architecture of OTCM is comprised of three main components: agent applications at the site of the customer where the industrial devices are installed, Insights Hub (which is a cloud-based platform of the case company, Siemens Oy), and the OTCM application. The directional arrows designate the data flow within the system. Firstly, the raw data regarding the health and performance of the installed machinery are collected by the servers at the different sites and are transmitted to Insights Hub which hosts the OTCM application and provides services such as the File API, Event Management API and Time Series for processing and manipulating data. The File API handles files, the Event Management API provides storage and management services for event-driven operations, and the Time Series service is utilised for processing and persisting time-dependent data sequences.

Secondly, the raw data is accessed, processed and analysed by the Event-processor of the OTCM application. Error event objects are then generated from the parsed event files (from the File API) and persisted to the cloud through the Event Management API. Additionally, an issue ticket object is automatically created and persisted to track each unique error event that occurred. Furthermore, the online status of the agent applications and the servers at the sites are parsed and updated. A subscription feature is also supported for notifying the maintenance team and interested operators or managers of the error events from the monitored machinery. The Event-processor is scheduled to run repeatedly at specific intervals, processing and persisting data to the cloud.

Thirdly, the created error events and issue tickets are accessed and further processed by the ‘Backend’ to generate a summary of events and other statistical data required for visualisation and analysis. In addition, aggregation endpoints are provided for interactions with the ‘Frontend’ of the OTCM application. Finally, the data is visualised in the ‘Frontend’. The data is accessed and translated into accessible visual formats, allowing maintenance personnel to monitor events, update the states of issue tickets and confirm the resolution of errors.

3.2 The Identified Problem

The performance of the OTCM application has been observed to degrade, with page loading times of thirty seconds which significantly exceeds the industry-standard threshold of two seconds, indicating inadequacy in the responsiveness of the application. The inefficiency in page loading was consistently noticed both in normal usage (characterised by one to five users) and in peak usage (exceeding five users) periods. Notably, this performance problem is isolated to web pages that interface with the Event Management API of the Insights Hub. Contrarily, other web pages or components of the OTCM application, which are not associated with the Event Management API, performed adequately. Consequently, the extended page loading times adversely impacted user experience, resulting in dissatisfaction and reduced productivity. Preliminary investigations excluded network latency as the origin of the performance problem. Instead, limitations such as request size constraints in the Event Management API, in addition to increased frequency of data retrieval and data processing inefficiencies in both the front-end and back-end of the application, are possible origins of the inadequate performance. This hypothesis is thoroughly investigated in the subsequent sections, using assessment tools such as DevTools and load testing with Locust, as discussed in the 'Performance Testing Tools' subsection. The objective is to identify specific bottlenecks and define strategies for optimising the OTCM application.

3.3 Performance Evaluation of OTCM

The performance evaluation of the OTCM application was conducted in two phases. Firstly, the efficiency of the web pages of the application was assessed with DevTools, and performance metrics such as TBT, TTFB and page loading time were collected and analysed. These recorded metrics in combination with the usage pattern of the application then guided the definition of the optimisation targets. Lastly, the Event Management API was load tested, and the results were analysed together with the web performance records to identify bottlenecks in OTCM.

3.3.1 Test Plan for Identifying Bottlenecks in OTCM

The following outlines the test plan for identifying bottlenecks in the OTCM application:

- The objective of this evaluation is to determine the response time and throughput of the Event Management API under varying load conditions, identify any performance bottlenecks specific to the API, evaluate the scalability of the API in handling concurrent requests, quantify the impact of the identified bottlenecks on user experience and establish a baseline for performance against which optimisation efforts are measured.
- The Testing environment is a Windows PC with thirty-two gigabytes of memory and a storage capacity of one terabyte.
- Loading performance of the OTCM application is recorded and evaluated with the Performance panel of DevTools. This involves monitoring JavaScript execution time to identify thread-blocking scripts or codes that execute for extended periods and identifying CPU-intensive tasks that may be affecting responsiveness.
- Analysis of the network activity of the web pages of OTCM with the Network panel of DevTools to inspect HTTP requests and record TTFB and resource loading times of the web pages.
- Reviewing the source code of OTCM to analyse the efficiency of the algorithms that are utilised, especially in data processing and retrieval functionalities, and evaluate the integration and usage of the Event Management API.
- Reviewing the documentation of the Event Management API to verify constraints specified by the API.
- Configuring Locust test environment and scripting tests to emulate various user interactions with the Event Management API.
- Stress testing the API (with a hundred and fifty users) by continuously increasing the load beyond the expected levels to measure the error rate and identify the breaking point of the system.
- Analysis of the locust test results together with the recorded web performance to identify bottlenecks and suggest optimisation strategies.
- Post-optimisation testing of OTCM to compare the performance of the improved OTCM to that of the current state.

3.3.2 Initial Performance Assessment with DevTools

The recorded performance of the Issue page of the OTCM is presented in Figure 7 below. The Issue page is the UI for visualising and managing issue tickets generated from error event objects.

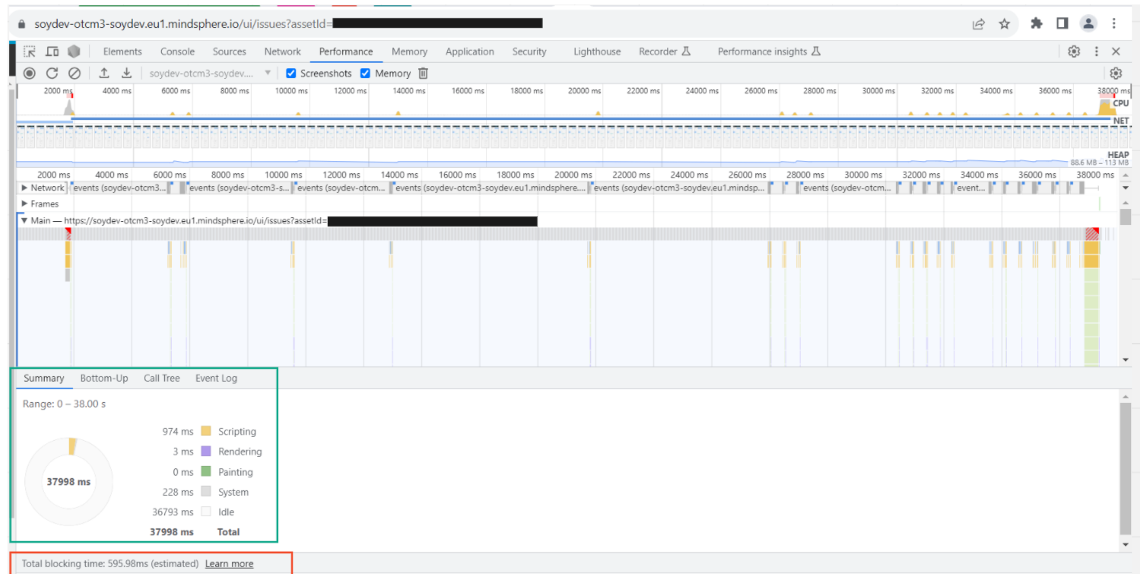

Figure 7. Loading Performance of the Issue Page of OTCM.

As depicted in Figure 7, a summary of the loading performance of the Issue Page is highlighted by the green rectangle at the bottom-left of the figure. The total loading time of thirty-eight seconds by the page significantly exceeds the acceptable benchmark. The rendering (three milliseconds) and the painting (zero milliseconds) times are minimal, indicating that the rendering and painting of the page elements are not a bottleneck. A significant portion (36793 milliseconds) of the total loading time (37998 milliseconds) was idle. This suggests a potential bottleneck related to network or script executing operations. Furthermore, the substantial scripting time of nine hundred seventy-four milliseconds and the recoded TBT (which is highlighted by the red rectangle at the bottom-left) of 595.98 milliseconds support the theory of a script execution bottleneck. Further analysis of the recorded performance of the page revealed a long task which significantly contributed to the TBT. A detailed view of Figure 7, showing the long task is presented in Figure 8 below.

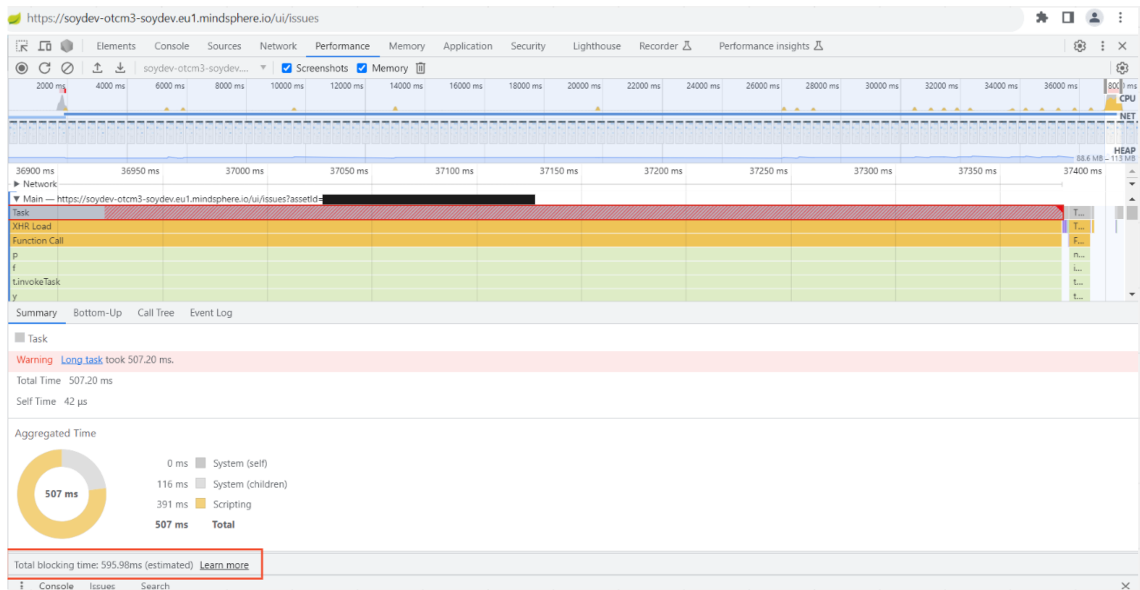

Figure 8. Long Task Recorded in the Loading Performance of the Issue Page of OTCM.

As illustrated in Figure 8, the long task, indicated by the red bar chart, was completed in a total of 507.20 milliseconds. This is a significant duration for a single task within web performance best practices, suggesting a performance bottleneck. The task prompted a warning by DevTools as the execution time of the task exceeded the acceptable limit of fifty milliseconds. Long tasks can block the main thread, leading to unresponsive pages.

The scripting time within the long task was further investigated by reviewing the source code to identify specific functions or lines of code that were causing the delays. Furthermore, the network activities of the page were inspected with the Network panel of DevTools to pinpoint the bottlenecks. The frontend script responsible for retrieving issue tickets (data) from the server (Event Management API) was pinpointed as the source of the TBT. The detected bottleneck is depicted in Listing 1 below.

```
do {
      if (issuesRequest) {
        const nextLink = issuesRequest._links.next?.href;

        issuesRequest = issuesRequest
          ? await lastValueFrom(
              this.http
                .get<EventsCollection>(this.baseUrl + nextLink)
                .pipe(
                  timeout(requestTimeout),
                  retry(numOfRetries),
                  catchError(
                    handleError(
                      this.notificationService,
                      'Cannot fetch issues!'
                    )
                  )
                )
            )
          : await this.getIssues(entityId, issueState, history);
      } else {
        issuesRequest = await this.getIssues(entityId, issueState, history);
      }

      issues = issuesRequest?._embedded
        ? [...issues, ...issuesRequest._embedded.events]
        : issues;
    } while (issuesRequest?._links ? issuesRequest._links.next : false);
    return issues;
  }
```

Listing 1. A TypeScript Loop for Retrieving Issue Objects from the Server.

Listing 1 presents a TypeScript function for fetching and aggregating issue event objects for the OTCM application. This function is executed during the loading of the Issue page. It is a loop that continuously fetches issues in batches from the Event Management API until all the issue objects have been retrieved. In each iteration of the loop, an asynchronous API call is executed which is dependent on the completion of the previous API call, potentially resulting in a long sequence of waiting times that decrease the user experience. The blocking time is much more significant when several batches (server pages) are being fetched. Additionally, the retry logic of the function could compound delays if the API is unresponsive or slow. These observations are further supported by the network activities of the Issue page which are depicted in Figure 9 below.

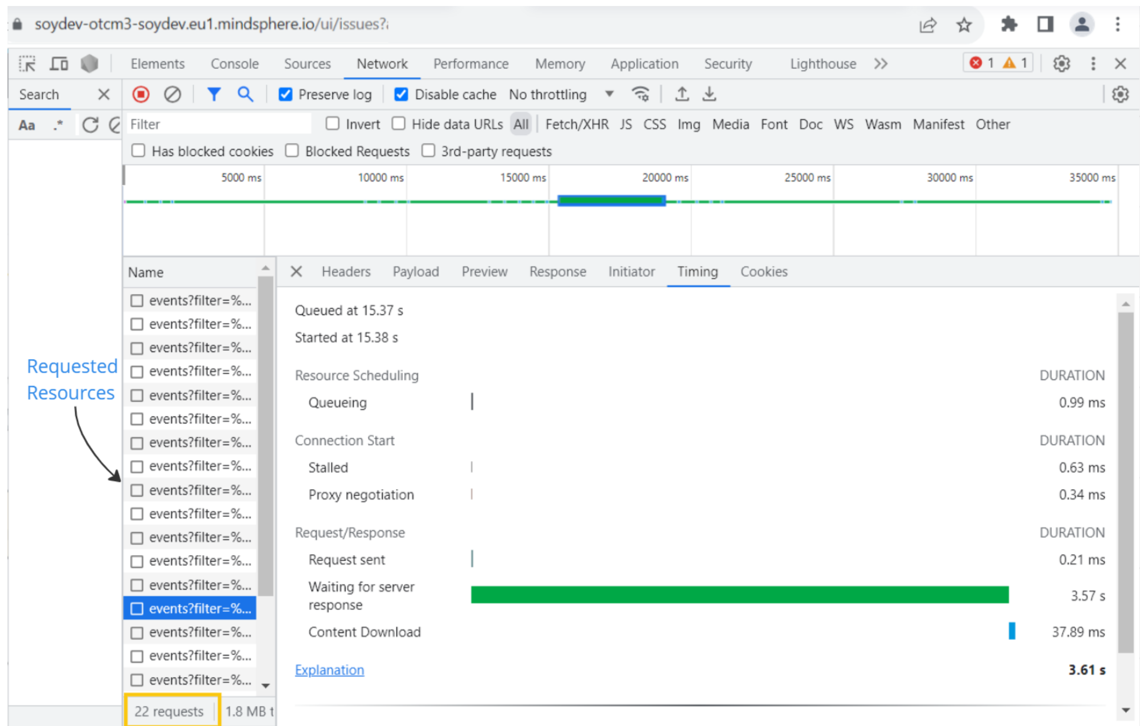

Figure 9. Network Inspection of the Issue Page of OTCM.

As presented in Figure 9, the network inspection of the Issue Page indicates that a sequence of twenty-two resources was requested from ‘eventsFilter’, the Event Management API. This activity corresponds to the expected output of the loop function depicted in Listing 1. The content download time for the selected request which is highlighted in blue is 37.89 milliseconds, which is an adequate duration and suggests that network transmission speed is probably not the cause of any performance issues. However, the TTFB of 3.57 seconds for the selected request is significantly large. The TTFB is indicated by the green bar chart. TTFB is the time it takes for the client to receive the first byte of data from the server after a request is initiated. High TTFB values generally signify server-side delays, which could involve slow database queries, resource-intensive computations, or other forms of server-processing bottlenecks. This suggests that the server response to the API call is a potential area to investigate for performance improvements, rather than the network conditions between the client and server.

3.3.3 Load Testing with Locust

The Event Management API was stress-tested to confirm the initial assessment that the API is the primary source of the bottlenecks (in the context of the OTCM application). The load testing and the simulated user behaviours were implemented using Locust, an open-source load testing tool. The detailed script, which includes tasks for fetching a list of events and handling the pagination of issue objects, is outlined in Appendix 1. The script is designed to simulate the long task (loop) depicted in Listing 1 above and measure the performance of typical user interactions under varying load conditions, particularly checking for response times increase, error rates, or any other performance degradation that occurs as the number of concurrent users grows. (see Appendix 1 for the Locust script). The results of the stress test are depicted in the charts in Figure 10 below.

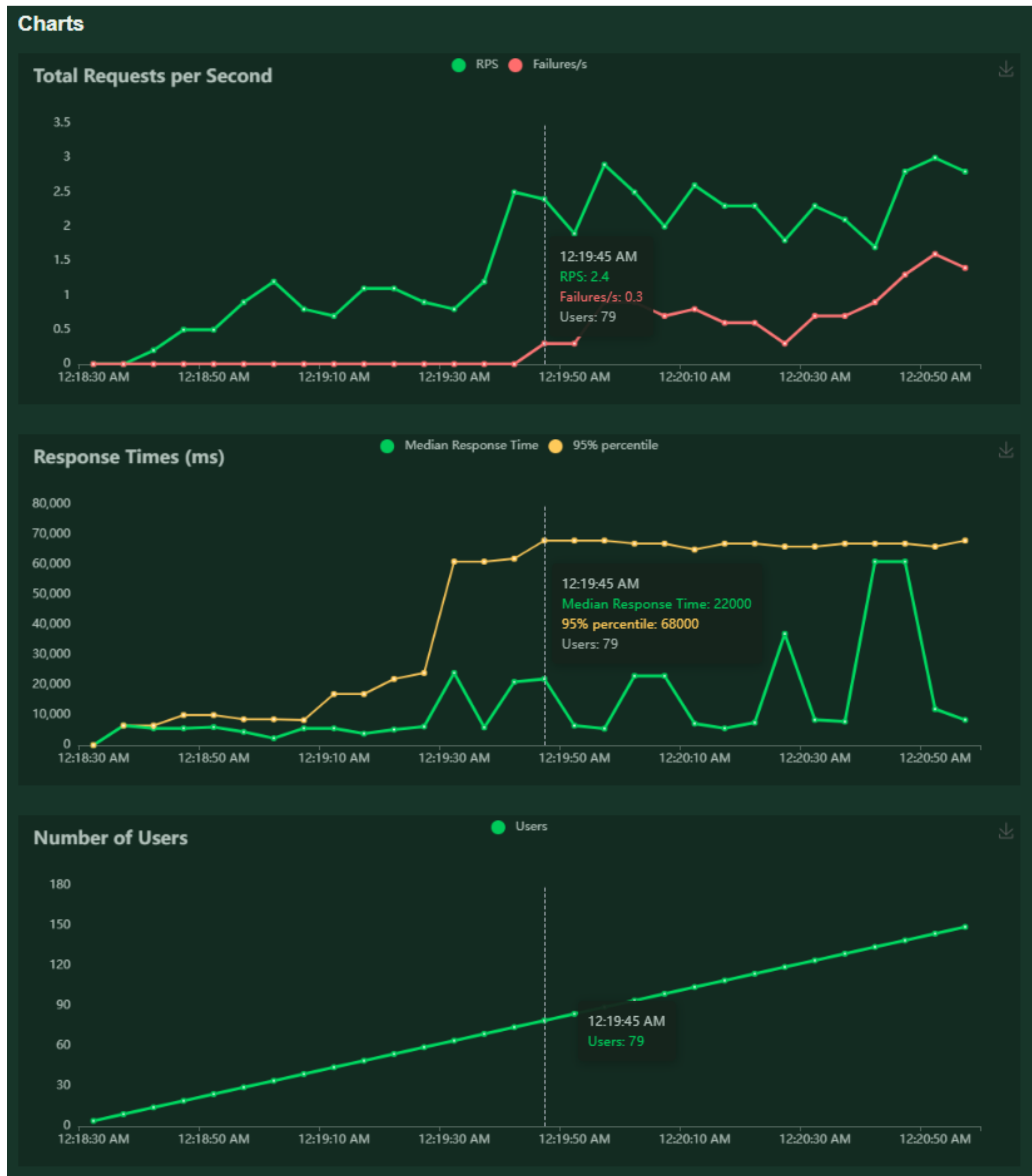

Figure 10. Recorded Performance of the Event Management API Usage in OTCM.

Figure 10 presents the results from a Locust stress test on the Event Management API, executed with the test script detailed in Appendix 1. The visual charts depict several critical performance metrics: the total requests per second (throughput), response times, and the number of users (load) over time. The Total Requests per Second chart at the top of the figure depicts two curves, the RPS (number of completed requests per second) in green and the red curve representing the error rate.

The RPS curve fluctuates with increasing loads but generally displays an ascending trend, peaking around 2.9 RPS. This demonstrates the ability of the API to handle an increased number of requests or users. However, the occurrence of errors with an increased load of over seventy-nine users implies that the API is not entirely stable under the test conditions. These could be due to server errors, resource constraints or timeouts. These results are correlated to the response time metrics depicted in the second chart of the figure. The median response time (green curve in the second chart) is the midpoint of the response times, it shows the time taken to complete fifty per cent of the requests. In contrast, the ninety-five percentile (the yellow curve) indicates that ninety-five per cent of the requests are completed with a response time below the displayed value. The median response time remained stable at around five seconds for a load below sixty users and spiked to sixty seconds after a load of a hundred and thirty users was applied. The spike correlates with the error rate curve, signifying a bottleneck. Conversely, for a load of thirty users, the ninety-five percentile was six seconds, implying that only five per cent of the requests took longer than six seconds to be completed. The ninety-five percentile increased to twenty seconds for a load of fifty users and spiked to a significantly large value, sixty-one seconds for seventy-nine concurrent users. Therefore, the API (as utilised in the OTCM) is stable but contains inefficiencies only observed under increased loads.
The error statistics recorded during the performance test are depicted in Figure 11 below.

Failures Statistics

Method	Name	Error	Occurrences
GET	/api/eventmanagement/v3/events?size=20&filter=%22typeId%22:%22soydev.sinema.server.event.v4%22,	500 Server Error: for url: https://gateway.eu1.mindsphere.io/api/eventmanagement/v3/events?size=20&filter=	23
GET	/api/eventmanagement/v3/events	500 Server Error: for url: https://gateway.eu1.mindsphere.io/api/eventmanagement/v3/events	24
GET	/api/eventmanagement/v3/events	504 Server Error: for url: https://gateway.eu1.mindsphere.io/api/eventmanagement/v3/events	10
GET	/api/eventmanag filter=%22typeId%22:%22soydev.sinema.OtcmIssue.v1%22,	504 Server Error: for url: https://gateway.eu1.mindsphere.io/api/eventmanagement/v3/events?filter=	15

Figure 11. Analysis of Server Error Rates During the Event Management API Stress Test.

As depicted in Figure 11, a total of seventy-two server errors occurred during the stress test. The ‘500 Server Errors’ are an indication of problems with server (API) stability or unhandled exceptions in the backend when handling a large number of requests. Conversely, the ‘504 Server Error’ (gateway timeout error) suggests that the API did not receive a timely response from an upstream server (possibly a database or another service the API is dependent on) before the request timed out, implying a bottleneck in the processing chain of the requests and reduced server responsiveness.

The stress test results presented in Figures 10 and 11 outline the performance of the Event Management API (as integrated into the OTCM) under varying load conditions. While the API maintained moderate stability at decreased user loads, the increase in response times and the occurrence of server errors as the load increased over time indicate performance bottlenecks. Specifically, the sudden increase in the ninety-five percentile of response times and the correlation with the increased error rates at extended user loads highlighted inefficiencies within the API.

The combined performance analyses of the current state of the OTCM application from the initial assessment with DevTools and source code review to the Locust stress test on the Event Management API, indicated a significant degradation in performance, particularly on the Issue page. The performance bottlenecks resulted from both server-side inefficiency (in the API) and complex data processing within the OTCM application. Though the API is relatively stable, the specific use cases of the API by the OTCM (such as requesting large amounts of data) are not optimised for performance by the Event Management API, thus leading to a suboptimal user experience characterised by long processing times and a substantial TBT. Therefore, optimising the OTCM and the Event Management API is imperative for improving performance and user experience. Potential strategies include database query optimisations, analysis of server logs to pinpoint the cause of the server errors, improving the logic for requesting and handling data, and increasing server resources.

Nonetheless, the Event Management API is external to the OTCM application, implying that the source code and the infrastructure of the API are not accessible to the developers of OTCM. Thus, it is impossible to directly address the underlying bottlenecks within the API to improve performance. Furthermore, it was observed that the integration of the Event Management API into the OTCM application was conducted following the ‘best practices’ of the API. This observation was conducted by reviewing the API documentation and the source code of the OTCM application. However, constraints such as rate limiting (limit to the number of requests per client to the API within a specific period) and server-side pagination defined by the API provider, necessitated implementations within the OTCM that impacted performance but were necessary to attain the required use cases of the application. For instance, server pagination limits data retrieval to a hundred records per request. Thus, a substantial number of requests are required for processing a large dataset (thousands of records). This results in inefficient consumption of the allowed request rate (rate limit), compounding to extended page loading times and reduced user experience. Therefore, the API is suboptimal for the specific use case of the OTCM which requires large data transfer. Consequently, the recommended optimisation strategy involves migrating the OTCM data and services from the Event Management API to a separate database service within the OTCM application and refactoring and optimising the application to utilise the new database.

3.3.4 OTCM Performance Goals

The following performance optimisation goals were defined for OTCM after the conducted performance analysis:

- The load time of the Issue page should not exceed three seconds.
- The load time of the Event page of the OTCM should not exceed two seconds.
- The API response time should not exceed one second for a load of twenty users.
- The error rate should be zero for load tests conducted on the API.

3.3.5 Recommended Optimisation Strategies

Considering the recommendation to replace the Event Management API with an internal database service, a series of strategic optimisations was proposed to address the identified bottlenecks and improve overall system efficiency and user experience:

- Migration to a dedicated internal database service: An internal OTCM database is implemented to transition data storage and processing from the external Event Management API to an internally managed PG cloud database, allowing for more flexible, efficient, and optimised data handling customised to the specific requirements of the OTCM application. A data migration plan should be designed to migrate existing data to the new system with minimal downtime, ensuring data integrity and consistency during the transition.
- Refactoring the OTCM application to optimise the interactions with the new database and moving intensive computational logic from the front-end to the back-end (of the OTCM) when necessary.
- Performance tuning of the new database: Regularly monitoring and optimising the new database performance through indexing, query optimisation, and adjusting configurations based on load patterns, ensuring the database is scalable to handle increased loads seamlessly.
- Implementation of a caching mechanism to store frequently accessed data and reduce response times.
- Load and stress testing to evaluate the performance of the OTCM application and the effectiveness of the implemented optimisations.

Thus, the limitations of the OTCM (resulting from the external Event management API) can be significantly mitigated by implementing these recommended strategies, ensuring improved performance, reduced load times and an enhanced user experience. These changes are crucial not only for immediate benefits but also to ensure the sustainability and adaptability of the system. An overview of the theory regarding the recommended strategies is provided in the following section.

4 Theory on Database Migration and Refactoring

The transition from an external API to an internal database service involves database migration and application refactoring. This section explores the theory of database, data migration and source code refactoring which is relevant in guiding the practical implementations described in the subsection, Recommended Optimisation Strategies.

4.1 Database and Database Migration

In the context of software, a database refers to a collection of data that is managed by a database management system (DBMS). The DBMS software system is designed to store, retrieve, manage, and manipulate data efficiently. In contrast, database migration is the process of transferring data from one database system to another. This usually involves both the physical data transfer and the database schema, ensuring the alignment of the source and the target databases with the technological and business requirements. [21, 22].

A typical database architecture includes several key components, each responsible for different database management features. The transport layer interacts with client applications and instances (nodes) within the database cluster, accepting requests, and ensuring that data requests and responses are accurately relayed. The received query is then transferred to the query processor for parsing, interpretation, and validation. Access control measures are also assessed at this stage to ensure compliance with security standards. The validated query is then progressed to the query optimiser to assess possible execution strategies for the query. This component is vital for determining the most efficient execution plan based on multiple criteria, including data location, index usage, and overall system statistics. The optimised plan is then executed by the execution engine which combines results from different operations to resolve the query effectively. The storage engine is responsible for core data manipulation operations such as data retrieval, creation, updating, and deletion, ensuring data integrity and consistency. [21, 23].

Thus, in a software context, a database is not limited to a repository of data, it is a complex system that includes hardware and software components designed to efficiently manage data operations across multiple nodes in a networked environment.

DBMS is categorised into key-value, columnar, document, graph, and relational databases, and each category is optimised for specific scenarios. Regarding key-value databases, data are stored as key-value pairs. This approach is simplistic but effective. This model is similar to ‘maps’ or ‘hash tables’ in programming, providing excellent performance and scalability for cases in which complex data relationships or queries are not required. Redis and DynamoDB stand out in this category, with Redis offering advanced data types and performance optimisation features. [24].

In Columnar databases, such as HBase and Cassandra, data is stored by columns instead of rows (as in relational databases). This orientation is advantageous for queries that predominantly access specific columns, allowing for efficient data compression and flexible schema design. These databases are particularly applicable for managing large-scale data warehouses where the agility in handling substantial amounts of sparse data is crucial. Columnar databases combine some relational database characteristics with the scalability of non-relational systems. [24].

Document-oriented databases such as MongoDB and CouchDB persist data as documents, which are structured as key-value pairs that can contain nested values. These systems are flexible, supporting variable data structures within documents and thus, preferable for applications requiring a dynamic schema. MongoDB is designed for scalability and consistency, while CouchDB provides compatibility across a wide range of environments from servers to mobile devices. [24].

In contrast, graph databases are designed to manage significantly interconnected data, representing entities as nodes and relationships as edges, each capable of containing additional properties. This category of DBMS is particularly beneficial to applications where relationships between data points (nodes) are crucial, such as in social networks or recommendation engines. Neo4J, a leading graph database, is capable of efficiently navigating complex relationship networks of interconnected nodes. [24].

Relational Database Management Systems (RDBMSs) represent the most conventional and universally adopted category of database management systems. RDBMSs are fundamentally based on set theory and structure data into two-dimensional tables, consisting of rows and columns, enabling a systematic and organised approach to data management. The primary tool for interacting with these systems is the Structured Query Language (SQL), which facilitates a wide range of operations including complex queries, updates, and general database management. The enforcement of data types and the ability to perform intricate relational operations are key strengths of the relational model. Examples of RDBMSs include PostgreSQL, MySQL, and SQLite. [24].

According to the theoretical perspective presented by Date (2019), an RDBMS consists of a collection of relation variables. Each relation variable (at a specific time) contains a relation value which is a set of tuples that accurately represent ‘true’ propositions as per the specific predicate of the relational model at a time. Thus, relation variables are dynamic and may change over time, but relation values are consistently maintained to conform to set theory principles. Each tuple in a relation variable embodies an instance of the predicate associated with that variable, and collectively, the tuples at any specific time represent a complete and accurate value of all ‘true’ propositions for that predicate at that moment. [25].

As affirmed by Dombrovskaya, Novikov and Bailliekova (2021), the query optimiser (optimisation engine) of PostgreSQL is considered the most efficient in comparison to the other DBMSs, making the database a preferred option to several expert database developers, and influencing the decision of it being utilised as the foundational code base for several commercial databases. Contrary to other DBMSs, execution plans for queries are not cached and reused by PostgreSQL. Instead, each query is optimised immediately before execution. This approach ensures that each query execution is optimised based on the current state of the database and the query parameters at that time. This methodology fosters more efficient data access and manipulation, improving response times. [23]. Therefore, in the context of this thesis, PostgreSQL is the preferred database for optimising the OTCM application.

4.1.1 Designing a Database

The quality of a database design is crucial in ensuring data integrity and adequate database performance. A suboptimal design may result in inefficient queries that cannot be resolved through query optimisation or indexing techniques. Therefore, investing time and effort into proper database design is essential to ensuring that the database can accurately and efficiently access data as required by the application. The designing process consists of three stages: requirements analysis, conceptual design, and logical design. In the requirements analysis phase, the focus is on understanding the objectives of the database, the types of data to be stored, and the relationships among those data elements. This is followed by the conceptual design stage, where the requirements are translated into an abstract model, outlining the main entities and entity-attribute relationships, typically visualised with Entity Relationship (ER) diagrams. The process concludes with the logical design, where the conceptual model is mapped onto a specific DBMS, transforming it into a detailed logical schema that specifies database tables, keys, and constraints, resolving data redundancy through normalisation. [23, 26, 27].

An ER diagram is a visual tool illustrating the relationships between database entities (such as students or courses). Each entity is typically represented as a rectangle. Relationships are depicted as lines connecting the entities, often with symbols indicating the nature of the relationship (such as one-to-many and many-to-many). Attributes, which are characteristics of an entity, detail the data structure and interrelations. ER diagrams are a crucial blueprint in the database design process, providing a clear and organised representation of data structure, which aids in communication and planning during development. [26, 27].

Data consistency and accuracy are partly achieved through the normalisation of the database design. Database normalisation is a mechanism in database schema design aimed at reducing data redundancy and improving data integrity. This process organises the database into multiple tables and establishes relationships between them through primary and foreign keys. The intent is to decompose a database such that each table achieves a certain normal form, with each form building on the previous one further reducing data dependencies and anomalies. The first three normal forms are most utilised in practice. The First Normal Form (1NF) requires that each table contain no repeating groups or composite attributes. Additionally, attribute values are unique, and a record is uniquely identifiable with a primary key. The second Normal Form (2NF) builds on the first by ensuring that all attributes in a table are completely dependent on the primary key, thereby minimising redundancy. This usually involves separating data into different tables connected with foreign key constraints. The third Normal Form (3NF) takes this further by ensuring that no column is dependent on other non-key attributes, eliminating transitive dependencies. [25, 26, 27]. Dombrovskaya, Novikov and Bailliekova (2021) further emphasised that the primary goal of normalisation is not to enhance performance but to prevent data inconsistencies and anomalies. Normalisation is achieved through the ER model by identifying relevant relations and eliminating repetitions. [23].

Therefore, effective database design is essential for maintaining data integrity and enhancing database performance. The careful analysis of application requirements, creation of ER models, and database normalisation ensure that databases are both efficient and reliable. This methodology ensures data quality and consistency in business operations.

4.1.2 Database Optimisation

Database Optimisation involves enhancing the performance of a database through strategic design, efficient query implementations, and continuous monitoring and assessment. This is achieved by applying indexing and query optimisation strategies to improve application efficiency. Additionally, it is recommended to include optimisation considerations both during the design and implementation stages of the database. [23, 27].

Indexing is the process of creating a data structure that improves the efficiency of data retrieval operations in a database. This data structure is stored as metadata in the database. Indexes are utilised in enhancing performance during data retrieval operations, especially in large datasets, by providing a quicker access path to the data compared to a full table scan. For instance, without an index, finding a specific record in a table might require sequentially scanning each row, which becomes increasingly inefficient as the size of the table (data) increases. Conversely, an index scan can significantly accelerate the search, reducing the number of rows required to be examined to a fraction of the total. In practice, when a query is processed by a DBMS, the decision to utilise indexes is automatically determined by the query optimiser based on whether an index scan would improve data retrieval speeds. Indexes are maintained automatically by the DBMS, which updates them in response to changes in the underlying table data. This maintenance can introduce some performance overhead, especially during frequent updates. However, the overall impact of indexes typically results in a net gain in performance, particularly in environments where read operations are more frequent than write operations. [21, 23, 27].

As a guideline for optimising database performance, indexes are created on foreign keys and columns that are frequently utilised in queries, provided that the speed of write operations is not adversely affected. Furthermore, the creation of indexes on non-discriminatory data, such as Boolean values that exist in one of two states, is not recommended. For example, fields limited to gender categories such as ‘male’ and ‘female’ are considered unsuitable for indexing, as the execution of queries on such data requires substantial processing time, thus, diminishing the potential benefits of indexing. [27].

Furthermore, understanding the different types of indexes is vital in ensuring that indexes are properly created to enhance query performance and data retrieval efficiency. Primary indexes are automatically created by the DBMS on the primary key column. This can significantly accelerate the retrieval of specific values. Secondary indexes, on the other hand, are used to improve performance for queries that involve non-primary columns. Composite indexes, which utilise multiple columns as a single index, are particularly useful for queries that test all the indexed columns concurrently. For scenarios where only a part of the table is relevant, partial indexes can be implemented. These indexes are particularly beneficial for queries with columns that contain many null values. Lastly, functional indexes are created based on expressions or functions applied to table columns, enhancing efficiency when queries are frequently searched by a transformation of the data. [23, 27].

The primary objective of query optimisation is to identify the most efficient method to execute a specific query by examining various execution strategies. An execution plan is a detailed blueprint outlining the query execution sequence by a database. Identifying the access methods such as table scans or index scans is vital for interpreting query execution plans. Additionally, the analysis should cover the types of joins, such as nested loops or hash joins, and their execution order, as these significantly influence the overall performance of the query. Understanding the order in which operations are executed is also crucial, particularly the placement of operations that minimise the dataset size early in the execution plan, which can greatly enhance performance. [23].

The following query-tuning techniques are essential in improving performance:

- Defining the correct indexes can significantly improve query performance but excessive or unnecessary indexes might degrade performance due to the overhead of maintaining them during write operations [23, 27].
- Writing efficient queries is another vital strategy. Subqueries should be replaced with 'join' or minimised to reduce execution time [23, 27].
- Using query hints provided by some DMBS to instruct the query optimiser to consider certain indexes or join methods over others [23, 27].
- Partitioning or dividing a table into smaller partitions can also improve query performance [23, 27].

Query optimisation requires knowledge of the data and the queries that interact with it. Proper use of indexes and assessment of execution plans can significantly enhance database performance. Continuous monitoring and tuning are essential in ensuring that the system continues to operate efficiently, adapting to different conditions. This continuous process is vital for maintaining the speed and effectiveness of a database system, ensuring a good user experience.

4.1.3 Data Integrity and Database Transaction

In DBMS, data integrity and consistency are ensured through a transaction. A database transaction is defined as a single and integral logical unit of work encompassing multiple operations, such as reading and writing database records. For a transaction to be successful, all the operations within the transaction are required to be executed successfully otherwise the transaction is rejected and rolled back. This concept is essential for maintaining the integrity and consistency of the database. Transactions are governed by four key properties, commonly referred to as ACID: Atomicity, Consistency, Isolation, and Durability. Atomicity ensures that the operations of a transaction are treated as all-or-nothing, implying that either all the operations are committed as a complete unit or entirely rolled back or reversed. [21, 23, 26].

A rolled-back transaction can be re-executed if it is necessary. Consistency ensures that during the execution of a transaction, the database is changed from one valid state to another, complying with database rules such as constraints and referential integrity. This property is partly user-defined, as it depends on the context of the application to determine a valid state. Isolation ensures that transactions are executed independently and transparently, without interference from other concurrent transactions. This is achieved through various isolation levels, which may compromise strict isolation to enhance performance. Finally, durability ensures that once a transaction is committed, the changes are permanently stored and are available to the application. [21, 23, 26].

The execution of transactions requires several key system components: The transaction manager coordinates the transactions, the lock manager manages access to resources, maintaining data integrity, the page cache stages change in memory before committing them to disk, and the log manager keeps a record of operations for recovery purposes. These components interact together to ensure the integrity and performance of the database during a transaction. [21].

4.1.4 Data and Schema Migration

Database migration is a critical process that can enhance scalability, improve performance, and optimise cost efficiency. It involves transferring data from one database system to another. This can include moving from an on-premises infrastructure to a cloud environment. This complex task requires careful planning and execution, ensuring data integrity, minimised downtime, and mitigation of potential risks. [22].

The initial phase of any database migration involves information gathering and assessment. This phase is crucial for understanding the specifics of the current database environment, including the data to be migrated, dependencies between schema objects, and key characteristics such as data volume and types. Tools such as the Data Migration Assistant (DMA) of Microsoft can automate this process, providing insights into compatibility problems, feature parity, and performance considerations between the source and the targeted environments. Based on business requirements and technical constraints, an appropriate migration strategy is selected. The 'Lift and Shift' (rehosting) strategy involves minimal changes to the database. In contrast, the refactoring strategy implements optimisations necessary for aligning with the capabilities of the new database environment. Conversely, rearchitecting is a complex strategy that requires significant modifications to fully utilise the technological benefits of the new platform. In the migration execution phase, the target database is configured according to the planned architecture and the data to be transferred. Techniques such as data replication, backup and restore or interim data storage solutions can be applied to ensure data integrity and minimise downtime. Post-migration, databases often require adjustments and tuning to optimise performance in the new environment. This could involve tuning SQL queries, restructuring indexes, or adjusting database configurations. Continuous monitoring is essential to identify any performance bottlenecks and optimise resource utilisation. Regular maintenance activities such as backups and updates are also vital for ensuring the health and availability of the database. [22].

To summarise, application performance can be improved through database migration. The process requires careful planning, effective use of migration tools and proper execution to ensure data integrity and minimise downtime. Whether through the Lift and Shift, Refactoring, or Rearchitecting strategies, each migration is defined to fulfil specific organisational and technical requirements.

In the context of this thesis, a custom migration script was implemented for migrating data from the external Event management API to a Microsoft Azure Database for PostgreSQL.

4.1.5 Migration with Knex

Knex.js is recognised for its versatility as a SQL query builder and migration tool that is designed to support various SQL databases such as PostgreSQL and MySQL, providing flexible and developer-friendly features that facilitate the building and execution of SQL queries. The tool allows for the programmatic construction of SQL queries using JavaScript features such as ‘promises,’ which ensure effective asynchronous flow control. Additionally, Knex offers features such as transaction query support, and connection pooling, ensuring standardised responses across different SQL dialects, and compatibility across various environments. As a migration tool, Knex facilitates the management of database schema changes through version-controlled migration files, ensuring consistent application across different development environments. Migration scripts can be executed with transactions, enabling migration rollbacks in cases of unsuccessful executions. Moreover, Knex supports database seeding which is a process crucial for populating databases with initial data, which is particularly beneficial during the development and testing phases. This ensures that all instances of an application start with a consistent dataset. The seeding process involves configuring the environment, creating seed files with the CLI of Knex, writing seed logic, and executing these seeds to populate database tables. [28].

4.2 Refactoring

Refactoring refers to the modification of existing source code to enhance readability and maintainability without altering the observable behaviour or functional output of the software. This process emphasises incremental changes that collectively optimise the codebase, facilitating future enhancements and bug fixes without disrupting the functionality of the software. Behaviour preservation ensures the observable functionality of the software is maintained through testing before and after changes are applied. Techniques such as renaming variables for clarity, breaking down large functions into smaller ones, and removing redundant code all contribute to a cleaner and more understandable codebase. [29].

Common refactoring techniques include the Extract Method, Rename Variable, Inline Temp, and Move Method. Each of these serves to enhance code modularity, reduce redundancy, and improve clarity, thus making the codebase more maintainable and less prone to errors. The Extract Method, for instance, enables the creation of modular, reusable code blocks, while the Move Method can reduce coupling between classes, enhancing cohesion within a class. Strategically timed refactoring can maximise benefits without disrupting the development workflow. Conversely, preparatory refactoring is conducted to prepare the code base for new functionalities. Comprehension refactoring is intended to familiarise a developer with a code. Regular refactoring is crucial for maintaining code quality, and facilitating agile development since developers become more familiar with the code base. [29].

5 Optimisation Target / Proposed Solution

Considering the OTCM optimisation strategies outlined in section three (Recommended Optimisation Strategies), and the Theory on Database Migration and Refactoring covered in section four, the optimisations depicted in Figure 12 below were proposed for the OTCM application.

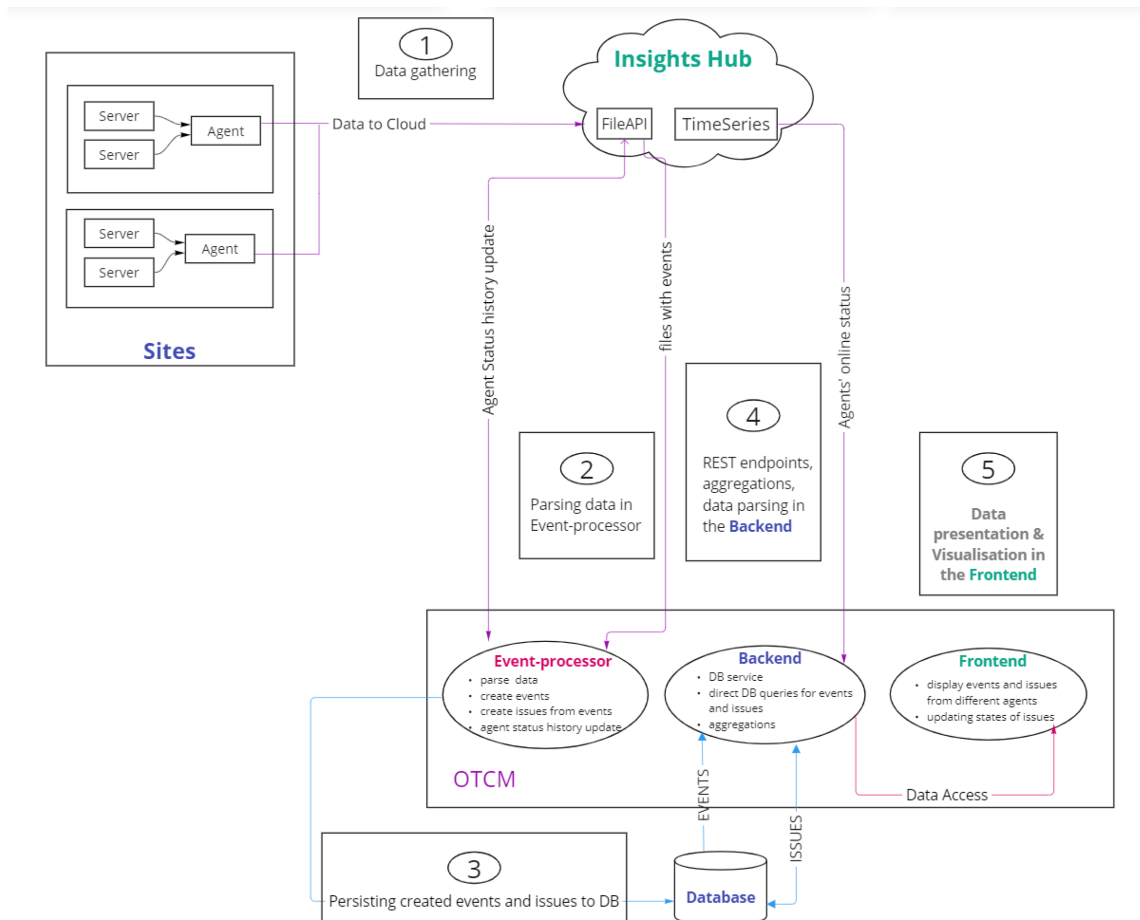

Figure 12. Optimised OTCM System Architecture.

Figure 12 depicts the system architecture of the optimised OTCM application. The optimised architecture is a result of strategic refactoring and migration efforts, aimed at addressing the bottlenecks identified in the initial system (Current State of OTCM) depicted in Figure 6. The original system depicted in Figure 6 depended on the external Event Management API for processing and storing data. This dependency indirectly introduced latency and complexity in data handling due to the nature of the external API calls.

In contrast, an internally controlled database is integrated into the optimised system (Figure 12). This change allows for tailored data management optimised for the specific requirements of the OTCM application, enhancing both data processing efficiency and storage capabilities. Furthermore, the Backend and Event-processor components of the optimised OTCM system are refactored to directly interact with the internal database, potentially improving the efficiency of data operations, and enhancing the overall system responsiveness.
The data model for the new internal database is illustrated by the ER diagram in Figure 13 below.

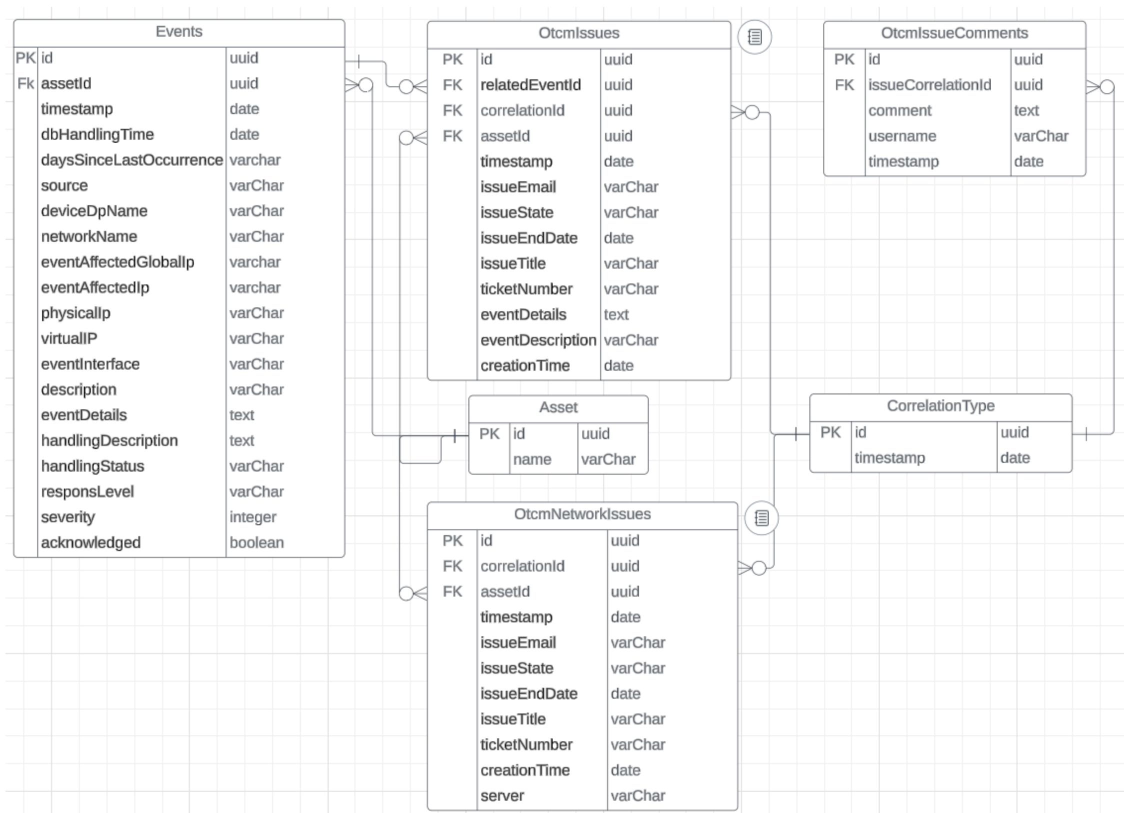

Figure 13. ER Diagram for the Optimised OTCM Database.

As depicted in Figure 13, the ER model of the OTCM database outlines the different entities (tables) in the database and the relationship among the entities. The Events table records every event detected by the OTCM system. It includes a unique identifier (id), reference (assetId) to the associated Asset, and details of each event recorded.

The Otcmissues table extends the functionality of the Events table by tracking issues that arise from the events. Each record includes an ‘id, a connection to the originating event (relatedEventId), and fields like issueState and issueEndDate that enable the management of the lifecycle of each issue. Additional fields like ‘ticketNumber’ enhance the tracking and resolution of issues. The Asset table maintains information on each asset monitored by the system, identified by a unique ‘id’ and name. This table connects assets to events and issues, allowing for asset-specific data analysis and management. Comments related to specific issues are stored in the OtcmissueComments table. It includes a unique ‘id’, and a reference (issueCorrelationId) to the related issue, ensuring proper documentation and traceability of the recorded issues. Network-related issues are stored on the OtcmissueNetworkIssues table. this table contains entries that are essential for diagnosing and resolving network-specific problems. Lastly, the CorrelationType table categorises the types of relationships or correlations between events and issues. It stores a unique ‘id’ for each type and a timestamp, allowing the system to group and record the history of issue tickets. The interconnections between tables are indicated by the lines representing the primary and foreign key relationships.

This model is designed to efficiently manage and query data regarding events, issues, assets, and their interrelations, which is essential for the functionality of the optimised OTCM system.

The proposed solution, illustrated in Figures 12 and 13, significantly enhances the efficiency and scalability of the OTCM application by transitioning from the external Event Management APIs to an internally managed database. This optimisation improved data handling and system responsiveness. The new ER model maps data relationships, enhancing management and analytical capabilities. Consequently, these improvements addressed the previously discussed bottlenecks and enhanced the performance of the system. The implementation of this solution is presented in the following section.

6 Implementation

The implementation of the proposed optimisations for the OTCM system was designed and executed in phases, ensuring the transition from the existing architecture to a new and more efficient system. This section outlines the execution process, from the initial setup to the performance testing of the optimised OTCM.

The first step involved cloning the existing source code of the OTCM into a separate development workspace to safeguard ongoing operations from any potential disruptions during the implementation process. This isolation of the development environment allowed for safe experimentation and testing without impacting the live system. The necessary development tools and packages such as the Knex migration tool were installed. Knex provides features for handling database schema migrations and seeding data, making it an ideal choice for managing database transformations efficiently.
Initially, the source code of the OTCM was cloned into a separate project workspace and the required packages such as the Knex migration tool were installed. A PostgreSQL database was also installed locally for development.

6.1 Migration of OTCM Database Schema

A new schema was created with Knex based on the ER model of the internal OTCM database, ensuring that all the data constraints were properly applied to support data integrity. Knex provided a structured and efficient means of handling schema migrations, ensuring that each change was properly versioned and that the integrity of the data was maintained throughout the transition process. The Implemented schema is presented in Figure 14 below.

```
dev2=> \dt
                   List of relations
 Schema |          Name           | Type  |  Owner
--------+-------------------------+-------+----------
 public | Assets                  | table | otcmtest
 public | CorrelationTypes        | table | otcmtest
 public | Events                  | table | otcmtest
 public | OtcmIssueComments       | table | otcmtest
 public | OtcmIssues              | table | otcmtest
 public | OtcmNetworkIssues       | table | otcmtest
 public | knex_migrations         | table | otcmtest
 public | knex_migrations_lock    | table | otcmtest
(8 rows)
```

Figure 14. Migrated OTCM Database Schema.

Figure 14 shows a screenshot of a PostgreSQL database command line interface, listing the relations (tables) in the migrated OTCM database schema. This listing is produced by the '\dt' command in a PostgreSQL environment, which displays the details of database tables such as schema name, table name, type, and owner. The 'Knex_migrations' and 'Knex_migrations_lock' tables are used by the Knex package for storing migration metadata. While the other tables in the list were derived from the ER model of the OTCM database.

6.2 Database Service Implementation

The database service was implemented using Knex as a query builder to facilitate the development process. The following performance considerations were applied: Indexing was strategically implemented on frequently accessed columns to improve query response times. The use of subqueries was minimised in favour of joins and partitions to reduce query complexity and execution time, particularly when multiple tables were involved. Connection pooling was implemented with Knex to ensure efficient connection management between the database server and the client. Finally, data integrity was implemented through PostgreSQL transactions, ensuring that data manipulations were processed reliably with changes being committed only if all parts of the transaction were successfully completed, thus avoiding partial data states. The project structure of the OTCM database service is illustrated in Figure 15 below.

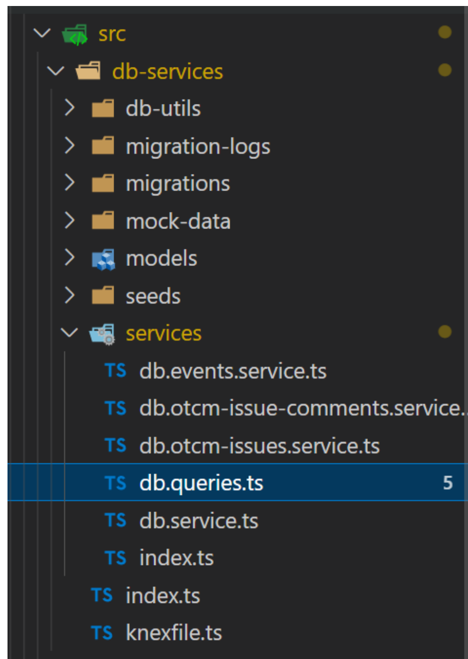

Figure 15. Project Structure of OTCM Database Service.

As depicted in Figure 15, the project structure consists of several components that interact with each other to provide the OTCM database service. Each component within the structure plays a critical role in ensuring the functionality and maintainability of the system. The ‘db-services’ folder contains TypeScript files that define the core services interacting with the database. These services are designed to encapsulate all database operations, promoting the separation of concerns and enhancing the modularity of the code. Adjacent to this, the ‘db-utils’ directory houses utility scripts and helper functions that assist various services in tasks such as database connection, error handling, and data formatting, which are pivotal for maintaining operational consistency across the application. the services directory encompasses files that handle more complex or aggregated data operations such as ‘db.events.service.ts’ for event-related data interactions, ‘db.otcm-issue-comments.service.ts’ for managing issue comments, and ‘db.otcm-issues.service.ts’ for issues handling. It also includes ‘db.queries.ts’ for centralising complex or frequently used queries and ‘db.service.ts’ for fundamental database operations, ensuring that all data management tasks are streamlined and accessible.

6.3 Data Migration

The data migration strategy was planned and executed using a custom script which is detailed in Appendix 2 (The script is depicted in Listing 1 of Appendix 2). During migration, the fetched data are validated and transformed by the ‘migrateDataFromMdspToDb’ function before being persisted into the database. This step was essential to ensure that the data conforms to the schema of the target database and to align it with the requirements of the system for data handling and storage. The transformation process includes reformatting data types, restructuring data arrays into database-friendly formats, and mapping data fields to correspond with database column names.

Furthermore, a logging mechanism was implemented throughout the migration process to record the details of each operation. This logging facilitated real-time monitoring of the migration process, allowing for quick identification and resolution of any issues that occurred. It also provided a data log that could be reviewed to verify the accuracy and completeness of the migration. An excerpt of the migration log is depicted in Listing 2 of Appendix 2.

6.4 Source Code Refactoring

The refactoring phase of the implementations was aimed at integrating the newly implemented database service and optimising the application logic to improve performance and maintainability, ensuring the efficiency of the system. The integration of the database was conducted incrementally to minimise disruptions and bugs. It began with the back-end services, proceeded to the Event-process and concluded with the front-end of the application. These incremental modifications and testing ensured that the proper functionality and behaviour of the application remained intact. In the front-end of the application, the code depicted in Listing 1 (which was indirectly responsible for a significant TBT and extended page loading times) was refactored into a simple function that executes a single API, thereby improving the performance of the application.

6.5 Performance Tuning

The refactored application with the fully integrated database service was further analysed and tuned for performance. The EXPLAIN command of PostgreSQL was utilised in analysing the execution of plans of different queries to identify areas for potential optimisation within the database operations. The EXPLAIN command in PostgreSQL outputs the execution plan for a query, outlining the sequence of executions required to retrieve or modify data. This insight is invaluable for diagnosing performance issues and optimising query execution paths. An example of such analysis is depicted in Figure 16 below.

Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

Sequence Scan

#	Node	Rows Plan
1.	→ Limit (cost=22945.15..22945.18 rows=15 width=394)	15
2.	→ Sort (cost=22945.15..23014.05 rows=27563 width=394)	27563
3.	→ Group (cost=19227.11..21993.27 rows=27563 width=394)	27563
4.	→ Gather Merge (cost=19227.11..21935.85 rows=22970 width=394)	22970
5.	→ Group (cost=18227.09..18284.52 rows=11485 width=394)	11485
6.	→ Sort (cost=18227.09..18255.8 rows=11485 width=394)	11485
7.	→ Seq Scan on Events as Events (cost=0..15965.07 rows=11485 width=394) Filter: (("timestamp" >= '2023-09-01'::date) AND ("timestamp" <= '2023-09-09'::date) AND ("assetId" = ███ uuid) A	11485

Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

Index Scan

#	Node	Rows Plan
1.	→ Limit (cost=15875.02..15875.05 rows=15 width=394)	15
2.	→ Sort (cost=15875.02..15943.97 rows=27581 width=394)	27581
3.	→ Group (cost=14784.62..14922.52 rows=27581 width=394)	27581
4.	→ Sort (cost=14784.62..14853.57 rows=27581 width=394)	27581
5.	→ Index Scan using idx_events_timestamp on Events as Events (cost=0.42..9180.33 rows=27581 width=394) Filter: (("assetId" = ███ uuid) AND ("responseLevel" = ANY ('{1,2,3}'::integer[])) AND (("handlingStatus")::text = ANY ('{not_m Index Cond: (("timestamp" >= '2023-09-01'::date) AND ("timestamp" <= '2023-09-09'::date))	27581

Figure 16. Execution Plans Illustrating Optimisation by Indexing.

As depicted in Figure 16, two execution plans are outlined for the same query to the Events table of the OTCM database. The top part of Figure 16 displays the execution plan for a sequence scan. This scan reads every row in the Events table to find the rows that fulfil the query condition. The plan details a multi-layered query process involving limits, sorts, group operations, and a gather merge, indicating a complex query involving multiple tables. The sequence scan is more costly and less efficient, especially with larger datasets, as it does not use any index to locate the data.

It is evident from the plan that sorting and grouping operations are particularly expensive in terms of computational resources, as indicated by the cost metrics. The total cost of operations for the plan with a sequence scan is approximately 23014.05, highlighting the considerable computational expense involved in processing without indexes. Conversely, the bottom section of the figure depicts an index scan, which utilises a predefined index on the Event table to directly locate the desired data, bypassing the need to scan every row. This method significantly enhances performance by reducing the computational load. The execution plan outlines similar sorting and grouping operations as in the sequence scan but at a substantially reduced cost totalling approximately 15943.79. The index scan leverages an index on ‘assetId’ and ‘timestamp’ conditions to efficiently filter and access data, minimising the rows processed and the overall execution cost. Thus, indexing benefits this query as it significantly improves computational efficiency. This analysis guided further optimisations across various queries within the application, improving overall responsiveness and efficiency.

6.6 Performance Testing of the Optimised OTCM

The performance of the optimised OTCM was systematically evaluated using the same benchmarking tools as in the initial assessments. The OTCM database service was stress-tested using Locust, while the web performance of the OTCM interface was assessed using Chrome DevTools. This approach ensured consistency in testing methodologies, directly comparing performance improvements.

The test environment mirrored the original setup, facilitating a controlled comparison of the optimised system against the original (current state of OTCM). The optimised OTCM was deployed in the same cloud environment, specifically on the Insights Hub, to replicate operational conditions. Additionally, an Azure Cloud Database for PostgreSQL was configured and utilised for the OTCM, where data previously managed by the Event Management API was migrated to this internal cloud database solution.

6.6.1 OTCM Database API Stress Testing

The first phase of performance testing focused on the database service. Stress tests were conducted to evaluate the resilience and capacity of the new OTCM database API. These tests aimed to validate the performance implementations that were conducted and to identify any potential bottlenecks in database handling and query performance that could impact the overall efficiency of the OTCM. The Locust test script for the OTCM database service is outlined in Listing 2 below.

```
class OTCMUser(HttpUser):
        wait_time = between(1, 3)
        host = host_url

        def on_start(self):
            self.setupAuth()
        @task
        def otcmEvents(self):
            try:
                response = self.client.post(
                base_url + events_url,
                json=EVENT_FILTER,
            )
                if response.ok:
                    fetchedEventList = (response.json()).get("events", [])
                    if len(fetchedEventList):
                        logging.info(f"Event list successfully fetched")
            except Exception as e:
                logging.error(f"Failed to fetch event objects: {e}")
        @task
        def otcmIssues(self):
            try:
                response = self.client.post(
                base_url + issues_url,
                json=ISSUE_FILTER,
            )
                if response.ok:
                    fetchedIssueList = (response.json()).get("issues", [])
                    if len(fetchedIssueList):
                        logging.info(f"Issue list successfully fetched")
            except Exception as e:
                logging.error(f"Failed to fetch issue objects: {e}")
```

Listing 2. Locust Load Testing Script for the Optimised OTCM

As depicted in Listing 2, the script for testing the optimised OTCM is similar to that for testing the original OTCM which is outlined in Appendix 1. The difference is in the targeted endpoints and the fact that the original script used a complex loop for requesting data.

In the original OTCM setup, the test script focused on interacting with the external Event Management API. In contrast, with the optimised OTCM, the script now targets internal APIs that communicate directly with the integrated PostgreSQL database, shifting the focus from external to internal database operations. This change enhances efficiency by leveraging direct control over data interactions, reflecting a significant architectural change in the system. The recorded performance of the stress test conducted is depicted in Figure 17 below.

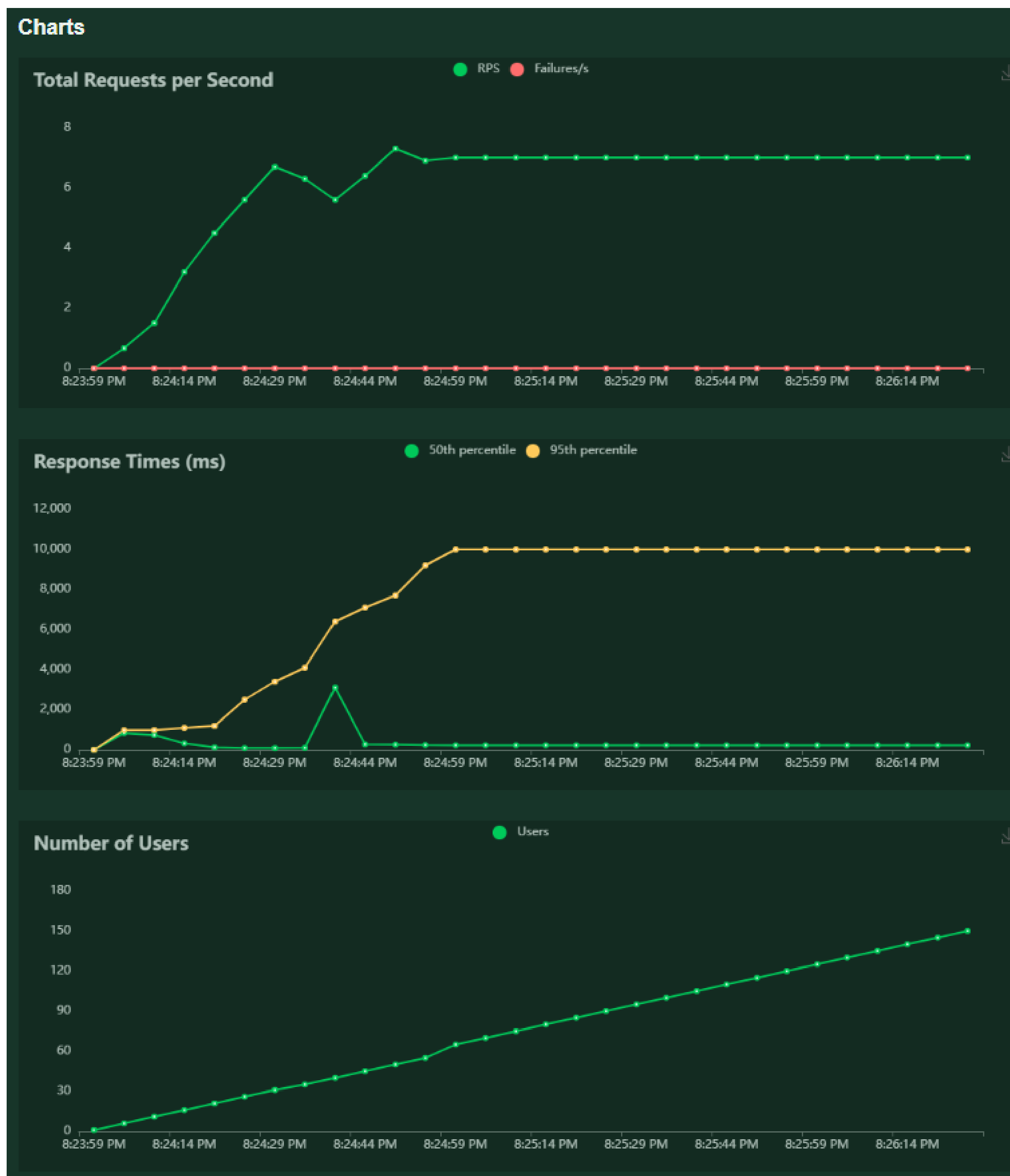

Figure 17. Performance Analysis of the Optimised OTCM Database Service.

As depicted in Figure 17, the performance analysis of the optimised OTCM system highlights several critical improvements over the performance of the original system, which is presented in Figures 10 and 11. The optimised version shows enhanced stability, scalability, and efficiency in handling increasing user loads and requests.

In the optimised OTCM, response times indicate significant improvement. Contrary to the original system, where the 95th percentile response times spiked dramatically to approximately sixty-eight seconds, the optimised system maintains much stable and lower response times, plateauing around ten seconds with increasing user loads. This stabilisation in response times under increasing loads suggests that the bottlenecks identified in the original system, particularly those related to database interactions and data processing, have been effectively addressed.

The throughput in the optimised system is also more stable and efficient. The Total RPS in Figure 17 consistently increased and plateaued at around 7.5 RPS without a corresponding increase in failure rates (no errors). This is a significant improvement over the original system, where throughput was erratic and failure rates increased with user load. The steady throughput in the optimised system indicates resilience and an enhanced ability to handle transactions efficiently.

Furthermore, the scalability of the system has been significantly enhanced. The user count in Figure 17 increases linearly without impacting system performance, which contrasts with the original system where increases in user load led to spikes in response times and failures. The optimised OTCM gracefully accommodates up to a hundred and fifty users, demonstrating the capacity of the system to manage increased loads efficiently. Error rates have also been reduced in the optimised system. The original system experienced frequent server errors during high-load scenarios, while the optimised system maintained performance with negligible failures. This reduction in errors is an indication of improved error-handling mechanisms and overall system resilience. This improved efficiency and responsiveness of the system are reflected in the web performance of the OTCM.

6.6.2 Web Performance Assessment

Following the database tests, the web interface of the OTCM was evaluated for page loading performance using DevTools. This was crucial to determine the effects of the backend improvements on the user experience, particularly regarding the response times and data rendering on the web pages. The web analysis of the Issue page of the OTCM is presented in Figure 18 below.

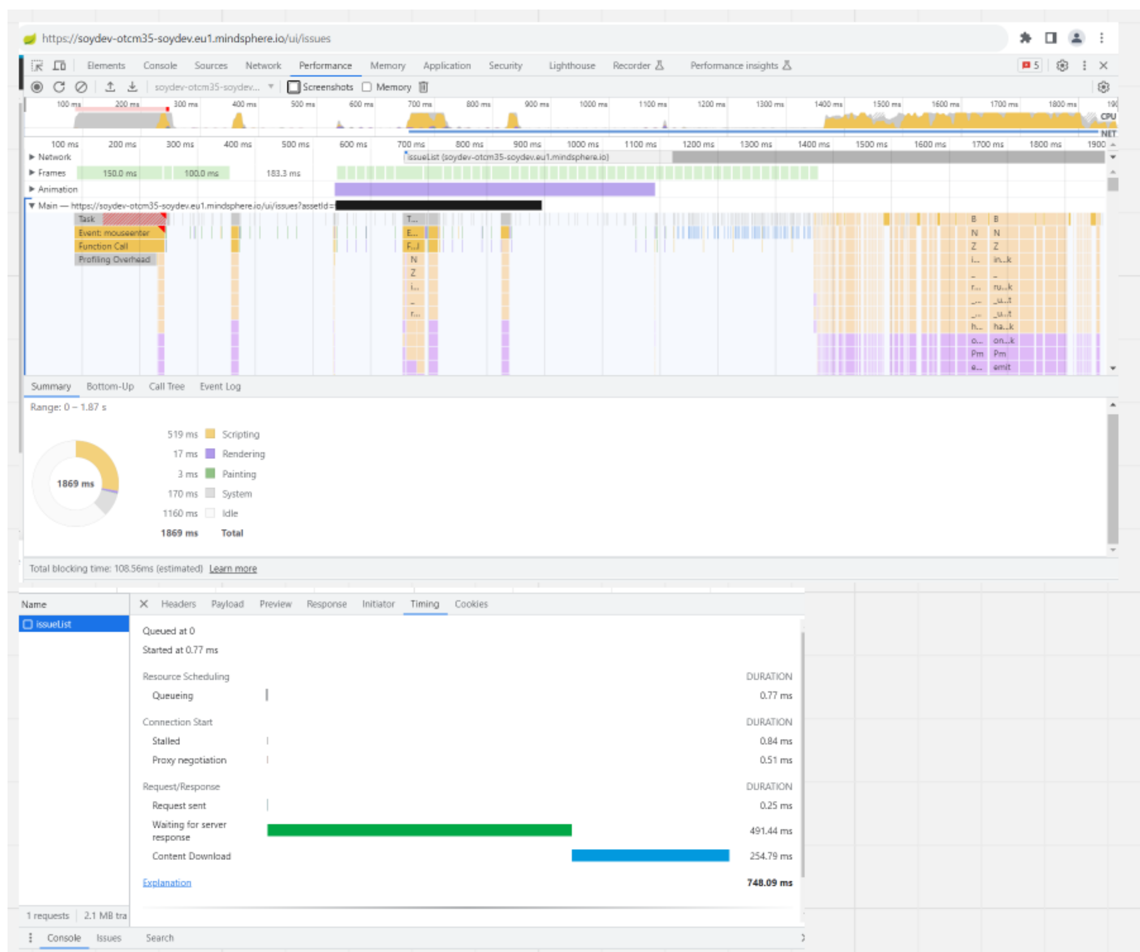

Figure 18. Web Performance Analysis of The Issue Page of the Optimised OTCM.

As illustrated in Figure 18, the web performance analysis of the optimised OTCM indicates significant improvements compared to the performance of the original OTCM presented in Figures 7, 8, and 9. In the original OTCM, the Issue page experienced prolonged loading times (with TBT of 595.98 milliseconds) and extensive periods where the browser remained idle during data fetch operations, resulting in total loading times of approximately thirty-eight seconds. This was largely due to inefficient script executions and excessive wait times for data fetched from an external Event Management API, which collectively contributed to the decreased responsiveness of the page.

In contrast, the performance metrics for the optimised OTCM depicted in Figure 18 indicate a drastically reduced page load time and a more efficient use of scripting and rendering phases. The page loads significantly faster, with a load time of under two seconds. This improvement is particularly noticeable in the decreased idle time, indicating efficiency in handling network requests and processing data.

These improvements are attributed to the refactoring of the front-end and the integration of the new internal database, which reduced dependencies on external data sources and optimised data retrieval and handling processes. The script responsible for fetching data has been streamlined, and network requests are handled more efficiently, as evidenced by decreased TBT and reduced wait times for server responses. This optimisation not only improved load times but also enhanced the responsiveness and user experience on the Issue page and other related pages of the OTCM application.

6.7 Summary of OTCM Performance Improvements

A summary of the recorded optimisations of the OTCM is presented in Table 1.

Table 1. Performance Comparison between the Original and Optimised OTCM.

Metric	Original OTCM (Pre-Optimisation)	Optimised OTCM (Post-Optimisation)
Total Requests per Second	gradually increased, peaking at around 2.9 RPS	steady increase, maintaining at around 7.5 RPS
Response Times (95th Percentile)	Spiked to 68 seconds	reduced and stabilised around 10 seconds
Median Response Times	stable at around 5 before spiking to 60 seconds at higher loads	remained stable at around 1 second even at higher loads
Number of Users	delays up to several seconds indicating server-side issues	Increased steadily to 150 users without performance drops
Failures per Second	occasional spikes correlating with increased load	none throughout testing
Total Blocking Time (TBT)	high, often contributing to slow page loads	Significantly reduced, enhancing page responsiveness
Page Load Time	could exceed 38 seconds in worst cases	Reduced to under 2 seconds consistently
Time to First Byte (TTFB)	delays up to several seconds indicating server-side issues (3.57 seconds)	significantly reduced, indicating a faster server response(0.491 seconds)

Table 1 presents a detailed comparison of performance metrics between the original and optimised OTCM systems. This table quantifies the improvements achieved through the optimisation efforts by comparing key indicators such as Total Blocking Time TBT, total RPS, RPS, and TTFB before and after the optimisations. The table illustrates the effectiveness of the refactoring and database migration strategies implemented, characterised by the reduced response times and increased efficiency in handling requests, thereby validating the successful enhancement of the performance and stability of the system.

7 Conclusion

This thesis aimed to enhance the performance and user experience of the OTCM application by migrating the database of the application from an external API to an internal PostgreSQL cloud database and refactoring the application for optimisation. The research and implementation focused on key aspects such as improving response times, reducing page load times, and handling larger user loads without degradation in performance. The migration to a PostgreSQL database and the subsequent refactoring of the OTCM application yielded significant improvements. Specifically, the load time for critical pages was significantly reduced from thirty-eight seconds to under two seconds, achieving the set performance goal and drastically enhancing the user experience. Nonetheless, continuous performance monitoring and tuning are recommended to ensure the maintainability and efficiency of the application.

While the thesis provides a comprehensive solution to identified challenges, future research could further explore the long-term effects of such migrations on system sustainability and cost-effectiveness. Additional studies could also assess the performance impacts of alternative cloud services and database systems to broaden the scope of optimisation strategies.

The outcomes of this thesis provide practical insights for software developers and system architects, demonstrating the effectiveness of cloud database migration and application refactoring in enhancing system performance. The project highlights the importance of a strategic approach to system design, particularly in the context of cloud technologies and large-scale applications. Thus, serving as a guide for future projects aimed at enhancing system performance and scalability.

References

1 Bierman G, Abadi M, Torgersen M. 2014. Understanding TypeScript. ECMA International.

2 Choi, David. 2020. Full-Stack React, TypeScript, and Node: Build cloud-ready web applications using React 17 with Hooks and GraphQL. Birmingham B3 2PB, UK: Packt Publishing Ltd. Electronic book. O'Reilly Online Learning. Accessed 1 October 2023.

3 Microsoft 2018. TypeScript: JavaScript with Syntax for Types. Microsoft Official Documentation.

4 Herron, David. 2018. Node.js Web Development: Server-Side Development with Node 10 Made Easy, 4th Edition. Birmingham B3 2PB, UK: Packt Publishing Ltd. Electronic book. O'Reilly Online Learning. Accessed 26 September 2023.

5 Node.js Foundation. 2020. About Node.js. Node.js Official Documentation.

6 Siemens 2023. Smart Solutions for Smart Cities. Siemens AG. Available from: https://plm.sw.siemens.com/en-US/insights-hub/

7 Siemens. 2023 Insights Hub - The Internet of Things (IoT) Solution. Siemens AG. Available from: https://developer.siemens.com/insights-hub/index.html

8 Myers, Glenford; Badgett, Tom & Sandler, Corey. 2011. The Art of Software Testing, 3rd Edition. Hoboken, NJ: John Wiley & Sons. Electronic book. O'Reilly Online Learning. Accessed 3 October 2023.

9 Liu, Henry H. 2009. Software Performance and Scalability: A Quantitative Approach. Hoboken, NJ: John Wiley & Sons. Electronic book. O'Reilly Online Learning. Accessed 3 October 2023.

10 Molyneaux, Ian. 2014. The Art of Application Performance Testing: From Strategy to Tools. 1005 Gravenstein Highway North Sebastopol, CA: O'Reilly Media, Inc. Electronic book. O'Reilly Online Learning. Accessed 4 October 2023.

11 Erinle, Bayo. 2017. Performance Testing with JMeter 3 - Third Edition. Birmingham B3 2PB, UK: Packt Publishing Ltd. Electronic book. O'Reilly Online Learning. Accessed 4 October 2023.

12 Wagner, Jeremy. 2016. Web Performance in Action: Building Fast Web Pages. Shelter Island, NY: Manning Publications. Electronic book. O'Reilly Online Learning. Accessed 4 October 2023.

13 Almeida, Virgilio; Dowdy, Lawrence & MenascÃƒÆ’Ã‚Â, Daniel. 2004. Performance by Design: Computer Capacity Planning by Example. Upper Saddle River, NJ: Pearson Education, Inc. Electronic book. O’Reilly Online Learning. Accessed 10 October 2023.

14 Transaction Processing Performance Council (TPC). 2010. TPC Benchmark. Available from: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. Accessed 10 October 2023.

15 Desikan, Srinivasan & Ramesh, Gopalaswamy. 2007. Software Testing: Principles and Practices. Delhi, India: Pearson India. Electronic book. O’Reilly Online Learning. Accessed 12 October 2023.

16 Locust. 2023. Locust Documentation. Online. Read the Docs. <https://docs.locust.io/en/stable/>. Accessed 30 October 2023.

17 Siddhant, Shrivastava & Prapulla, SB. 2020. Comprehensive Review of Load Testing Tools. International Research Journal of Engineering and Technology (IRJET), May 2020, pp. 3392 - 3395. IRJET. <https://www.irjet.net/archives/V7/i5/IRJET-V7I5651.pdf>. Accessed 30 October 2023.

18 Google Chrome. 2016. Chrome DevTools Documentation. Online. Google Chrome. < https://developer.chrome.com/docs/devtools/>. Accessed 11 November 2023.

19 Basques, Kaycey & Emelianova, Sofia. 2017. Performance Features Reference. Online. Google Chrome Documentation. <https://developer.chrome.com/docs/devtools/performance/reference >. Accessed 11 November 2023.

20 Basques, Kaycey & Emelianova, Sofia. 2015. Inspect Network Activity. Online. Google Chrome Documentation. <https://developer.chrome.com/docs/devtools/network/>. Accessed 11 November 2023.

21 Petrov, Alex. 2019. Database Internals. Sebastopol, CA: O’Reilly Media, Inc. Electronic book. O’Reilly Online Learning. Accessed 10 April 2024.

22 Kline, Kevin; McDowell, Denis; Dorsey, Dustin & Gordon, Matt. 2022. Pro Database Migration to Azure: Data Modernization for the Enterprise. New York City, NY: Apress. Electronic book. O’Reilly Online Learning. Accessed 16 April 2024.

23 Dombrovskaya, Henrietta; Novikov, Boris & Bailliekova, Anna. 2021. PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries. New York City, NY: Apress. Electronic book. O’Reilly Online Learning. Accessed 15 April 2024.

24 Perkins, Luc; Redmond, Eric & Wilson, Jim. 2018. Seven Databases in Seven Weeks, 2nd Edition. Raleigh, NC: Pragmatic Bookshelf. Electronic book. O’Reilly Online Learning. Accessed 15 April 2024.

25 Date, C.J. 2019. Database and Relational Theory: Normal Forms and All That Jazz. New York City, NY: Apress. Electronic book. O’Reilly Online Learning. Accessed 15 April 2024.

26 Grippa, Vinicius M. & Kuzmichev, Sergey. 2021. Learning MySQL, 2nd Edition. Sebastopol, CA: O’Reilly Media, Inc. Electronic book. O’Reilly Online Learning. Accessed 16 April 2024.

27 Harrington, Jan L. 2016. Relational Database Design and Implementation, 4th Edition. Burlington, MA: Morgan Kaufmann. Electronic book. O’Reilly Online Learning. Accessed 16 April 2024.

28 Knex.js. 2023. Knex.js Documentation. Online. Knex.js. <https://developer.chrome.com/docs/devtools/>. Accessed 19 April 2024.

29 Fowler, Martin. 2018. Refactoring: Improving the Design of Existing Code. Boston, MA: Addison-Wesley Professional. Electronic book. O’Reilly Online Learning. Accessed 20 April 2024.

Locust Load Testing Scripts for OTCM

This appendix includes a Locust script (written in Python programming language) for load testing the Event Management API of Insights Hub within the context of the OTCM application. The script depicted in Listing 1 below is designed to simulate user behaviour by executing HTTP requests to fetch event objects and issue lists, reflecting the actual usage patterns of OTCM.

```
class OTCMUser(HttpUser):
        wait_time = between(1, 3) # User wait time between tasks
        host = host_url

        def on_start(self):
            self.setupAuth()
        @task
        def eventObjectsSelected(self):
            # Simulate fetching a list of event objects
            try:
                response = self.client.get(event_url)
                if response.ok:
                    fetchedEventList    =    (response.json().get("_embedded",
{})).get("events", [])
                    if len(fetchedEventList):
                        logging.info(f"Event list successfully fetched")
            except Exception as e:
                logging.error(f"Failed to fetch event objects: {e}")
        @task
        def otcmIssues(self):
            otcmIssueList = []
            page = 0
            # Simulate fetching a list of issues with pagination
            while True:
                try:
                    issueUrl = setIssueUrl(page)
                    response = self.client.get(issueUrl)
                    if not response.ok:
                        logging.error(f"Failed to fetch page {page} of issues:
{response.status_code}")
                        break
                    issues = response.json().get("_embedded", {}).get("events",
[])
                    otcmIssueList.extend(issues)
                    logging.info(f"Fetched  {len(issues)}  issues  from  page
{page}")
                    totalPages = response.json().get("totalPages", 0)
                    if page >= totalPages - 1:
                        break
                    page += 1
                except Exception as e:
                    logging.error(f"Exception  during  pagination  of  issues:
{e}")
            logging.info(f"Total issues fetched: {len(otcmIssueList)}")
```

Listing 1. Locust Load Testing Script for OTCM.

As depicted in Listing 1 above, the ‘OTCMUser’ class defines a simulated user interacting with the Event Management API as integrated into the OTCM application. Two primary tasks are defined to emulate distinct user interactions as part of the testing process. The first task, ‘eventObjectsSelected’ is responsible for making GET requests to retrieve event data from a specified ‘event_url’. The second task, ‘otcmIssues’ is built to handle the pagination process, iteratively fetching issue data starting from page zero and appending each batch to an ‘otcmIssueList’ until all pages have been processed. This task was designed to emulate the data retrieval logic that is executed when the Issue page of the OTCM application is loaded. The script is executed through the Locust command-line interface, and the recorded performance is represented as charts and statistics by the Locust testing tool.

Data Migration Script for OTCM Database

This appendix outlines the data migration strategy used in transferring data from Insights Hub to the OTCM database. The migration script is depicted in Listing 1 below.

```
export async function seed(knex: Knex): Promise<void> {
  const assets = (await getAssetsFromMindSphere()).map((asset) => {
    return { id: asset.assetId, name: asset.name };
  });
  await knex(DbRelations.assets).insert(assets).onConflict('id').ignore();
  let isNotCompleted = true;
  const before = new Date();
  let fromTimestamp: Date | undefined = undefined;
  let toTimestamp = new Date();
  while (isNotCompleted) {
    await migrateDataFromMdspToDb(
      knex,
      OtcmTypeIds.event,
      DbRelations.events,
      false,
      toTimestamp,
      fromTimestamp
    );
    await migrateDataFromMdspToDb(
      knex,
      OtcmTypeIds.issue,
      DbRelations.issues,
      true,
      toTimestamp,
      fromTimestamp
    );
    await migrateDataFromMdspToDb(
      knex,
      OtcmTypeIds.comment,
      DbRelations.comments,
      true,
      toTimestamp,
      fromTimestamp
    );
    fromTimestamp = toTimestamp;
    toTimestamp = new Date();
    const timeDifference = Math.round(
      (toTimestamp.getTime() - fromTimestamp.getTime()) / (60 * 1000)
    );
    isNotCompleted = timeDifference > 5;
  }
  logger.shutdown();
}
```

Listing 1. Data Migration Script for OTCM Database.

Listing 1 presents a data migration routine implemented using Knex.js, a flexible SQL query builder for Node.js. This script is particularly tailored for the optimised OTCM system, ensuring a seamless and efficient migration of data from the Event Management API of Insights Hub (MindSphere) to the new internal PostgreSQL database for OTCM. This script is structured to perform both initial data seeding and data migration, ensuring that the system remains updated with the latest data from the external API.

The migration process begins with fetching asset data from MindSphere using the getAssetsFromMindSphere function, which interacts with the MindSphere API to retrieve a list of assets. Each asset retrieved includes an ‘id’ and a name, which are then mapped to a new object format suitable for database insertion. This transformed data is subsequently inserted into the assets table of the database using the insert method of Knex. To handle potential conflicts, such as duplicate entries, the script employs the ‘onConflict('id').ignore()’ method, which effectively skips the insertion of any asset that already exists in the database, thus avoiding duplicate key errors.

Following the initial seeding of asset data, the script implements a migration loop intended to transfer all the available data (events, issues, and comments) from Insights Hub to the OTCM database. This loop uses two timestamps, fromTimestamp and toTimestamp, to define the time range for each data fetch operation. The loop initiates without a fromTimestamp to capture all available data up to the current moment for the first iteration. In subsequent iterations, fromTimestamp is updated to the timestamp of the previous fetch (toTimestamp), and toTimestamp is set to the current time, ensuring that only new data is fetched and migrated.

Within each iteration of the loop, the migrateDataFromMdspToDb function is called three times to handle different data types. Each function call specifies the type of data to be migrated, the corresponding database table, and a boolean indicating whether the migration should account for the time range specified by the timestamps. The script is designed to continue these operations until the time difference between the fromTimestamp and toTimestamp is less than or equal to five minutes, a condition that helps manage the load on both the MindSphere API and the database, ensuring efficient and timely data updates.

During migration, the fetched data are validated and transformed by the 'migrateDataFromMdspToDb' function before being inserted into the database. In addition, database write operations were conducted in transactions to ensure the integrity of the migrated data. Logging was also implemented in the data migration process to track and monitor the migration. A sample log from the migration is outlined in Listing 2 below.

```
[32m[2023-09-07T09:37:04.853] [INFO ] [DbMigrationLogger] - [39m{
    batches: {
        [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 224482,
        'Total Events in batch': 224482,
        'current page': 2244,
        'Total pages in batch': 2245,
        'Total elements from this batch inserted into DB': 223985
      },
      [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 0,
        'Total Events in batch': 0,
        'current page': 0,
        'Total pages in batch': 0,
        'Total elements from this batch inserted into DB': 0
      },
      [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 0,
        'Total Events in batch': 0,
        'current page': 0,
        'Total pages in batch': 0,
        'Total elements from this batch inserted into DB': 0
      },
      [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 0,
        'Total Events in batch': 0,
        'current page': 0,
        'Total pages in batch': 0,
        'Total elements from this batch inserted into DB': 0
      },
      [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 1987,
        'Total Events in batch': 1987,
        'current page': 19,
        'Total pages in batch': 20,
        'Total elements from this batch inserted into DB': 1000
      },
      [assetName]: {
        'Current Asset': 'redacted',
        'Total Fetched from this asset': 2831,
        'Total Events in batch': 2831,
        'current page': 28,
        'Total pages in batch': 29,
        'Total elements from this batch inserted into DB': 2000
      }
    },
    extraInfo: {
      'Total Events Fetched from all Assets': 229300,
      "Total elements of type 'redacted' from all batches inserted": 226985,
      "Total CorrelationIds of type 'redacted' from all batches inserted": 0,
      'Total RelatedEventIds with missing or Non-existing Event in Mdsp': 0,
      'filter; timestamp; after:': '2023-09-06T23:51:29.194Z'
    }
  }
```


Listing 2. Data Migration Log Excerpt with Redactions.

Listing 2 depicts an excerpt of the data migration log created during the migration phase of the OTCM optimisation:

- Asset-Specific Data: Logs details for each asset including the number of events fetched, processed, and successfully inserted into the database.
- Pagination and Batch Processing: Tracks the pagination process showing the current and total pages, which is crucial for managing large datasets.
- Aggregated Data: Provides a summary of all events processed, including totals for different types of data elements and any issues with ‘correlationIds’ IDs or ‘eventIds’.
- Timestamps for Data Fetching: Indicates the specific timestamps used to filter data fetched from the source system, ensuring data consistency and completeness.

The structured logging format ensures that the information regarding the migration process is recorded and readily available for review. This supports ongoing monitoring, quick identification of issues, and the ability to perform detailed audits of the data migration process.