Amal Kayed

Improving Quality Assurance by Providing Robust Tools

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

8 May 2023

Abstract

Author:	Amal Kayed
Title:	Improving Quality Assurance by Providing Robust Tools
Number of Pages:	55 pages
Date:	8 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Ilpo Kuivanen, Senior Lecturer

In today's software industry, the success of any product is inextricably linked to its quality. Consequently, quality assurance has become a fundamental process in the development of high-quality software that meets customer requirements and expectations. The primary objective of the present study thesis was to leverage the powerful features and characteristics of robust tools to detect and manage defects that may arise during software development. The ultimate objective is to minimize risks and vulnerabilities as early as possible in the development cycle, thereby ensuring that a top-quality product is delivered to the client while avoiding any waste of valuable resources, such as time, money, and effort.

The study concentrated on Keikkakaveri, a web application that faced significant issues due to the lack of quality assurance in its early development. To enhance the application's robustness and mitigate potential security flaws, the research was carried out in three methodically planned phases, each focusing on addressing a specific issue that affected the product. The precise selection of suitable tools, aligned with the objectives of the study, project requirements, and technology stack employed in the development process, played a pivotal role in successfully resolving each problem.

To ensure successful software development, quality assurance plays a crucial role in delivering high-quality and secure software that meets customer requirements. By prioritizing quality assurance in development schedules and conducting meticulous monitoring and analysis throughout the development cycle, optimal product performance and quality can be achieved.

Keywords: Quality assurance, static analysis, static code analysis, typescript, unit test, integration test

Tiivistelmä

Tekijä:	Amal Kayed
Otsikko:	Tehokkaat työkalut koodin laadun parantamiseksi
Sivumäärä:	55 sivua
Aika:	08.05.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Ilpo Kuivanen

Nykyään ohjelmistoalan menestys on kiinteästi sidoksissa tuotteen laatuun. Laadunvarmistus on näin ollen keskeinen prosessi korkealaatuisen ohjelmiston kehityksessä, joka vastaa asiakkaan tarpeita ja odotuksia. Tämän insinöörityön keskeinen tavoite on hyödyntää vahvoja ominaisuuksia ja piirteitä vahvojen työkalujen avulla, jotta voidaan havaita ja hallita ohjelmistokehityksen aikana mahdollisesti ilmeneviä virheitä. Päämääränä on minimoida riskejä ja haavoittuvuuksia jo kehityssyklin alkuvaiheessa, jotta voidaan toimittaa korkealaatuinen tuote asiakkaalle välttäen arvokkaiden resurssien, kuten ajan, rahan ja vaivan, tuhlausta.

Tutkimus keskittyi Keikkakaveriin, verkkosovellukseen, joka kohtasi merkittäviä ongelmia riittämättömän laadunvarmistuksen vuoksi sen varhaisessa kehitysvaiheessa. Sovelluksen vahvistamiseksi ja mahdollisten tietoturvaongelmien lieventämiseksi tutkimus suoritettiin kolmessa systemaattisesti suunnitellussa vaiheessa, joista jokainen keskittyi tiettyyn tuotetta koskevaan ongelmaan. Oikeiden työkalujen tarkka valinta, jotka olivat yhteneväisiä tutkimuksen tavoitteiden, projektivaatimusten ja kehitysprosessissa käytetyn teknologiapinon kanssa, oli ratkaisevan tärkeää ongelmien onnistuneessa ratkaisemisessa.

Varmistaaksesi onnistuneen ohjelmistokehityksen, laadunvarmistus on ratkaisevan tärkeä tekijä, joka takaa korkealaatuisen ja turvallisen ohjelmiston, joka vastaa asiakkaan tarpeita. Laadunvarmistuksen priorisoiminen kehitysaikataulussa sekä tarkkaavainen seuranta ja analyysi koko kehitysvaiheen ajan varmistavat optimaalisen tuotteen suorituskyvyn ja laadun.

Avainsanat: Laadunvarmistus, staattinen analyysi, staattinen koodianalyysi, typescript, yksikkötesti, integrointitesti

Contents

Abbreviations

AST:	Abstract Syntax Tree. The converted TypeScript source code by TypeScript ESLint Parser.
API	Application Programming Interface. APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.
DAST	Dynamic application security testing. Dynamic code analysis tool.
VS Code	Visual Studio Code. A streamlined code editor with support for development operations like debugging, task running, and version control.
OAT	Operational Acceptance Testing. Evaluates whether a system or application is ready for real-world use by testing its performance, reliability, and functionality in typical usage conditions before release.
UAT	User Acceptance Testing. Assesses whether a system or application meets end-users' needs by testing its functionality, usability, and compatibility in a simulated real-world environment before release.
IDE	Integrated Development Environment is a software application that provides a comprehensive set of tools and features for developing software, including source code editing, debugging, testing, and building. IDEs are used to streamline the development process and improve productivity.
GitHub	Cloud-based platform for version control and collaboration in software development, built on Git. It provides a centralized repository for storing and managing code, enabling efficient teamwork and open-source contributions.

1 Introduction

The ownership of the project is held by Metropolia University of Applied Sciences, and it represents a sophisticated web-based application that efficiently manages and organizes daily business relationships between three crucial parties: the employee, business company, and agency company. Its overarching mission is to promote and enhance occupational safety, occupational health, and overall well-being.

To achieve this, the project is technically focused on leveraging the benefits of an electronic document management system. This will enable seamless document storage, retrieval, management, and utilization, in a manner that is both efficient and effective.

In terms of development, the project has been expertly crafted with ReactJS and TypeScript, which serve as the bedrock of the programming language. The client-side tools integrated into the project include React, Redux, Formic, Yup, and Material UI, while the server-side tools are comprised of NodeJS, Express, and MongoDB.

The study at hand has been meticulously crafted with a keen eye for details and unwavering dedication, buoyed by the sincere hope that this thesis will serve as a catalyst for innovation in the field. With a focus on improving business relationships for all stakeholders involved, this thesis is sure to be a valuable resource and a source of inspiration for future research.

1.1 Problem Description

During the development of the project, the need arose to change the development team before the factual development was complete, resulting in the involvement of multiple developers working on the same project for brief periods. Unfortunately, the frequent reorganization of software teams caused chaos and

negatively impacted the project's quality assurance. The resulting inconsistent code style within the teams made the code unreadable and difficult to maintain, leading to delays in the project timeline. Furthermore, the developers who joined the team for brief periods had an incomplete understanding of the software architecture, resulting in an increase in software bugs, errors, and security vulnerabilities that compromised the integrity of the software. These challenges emphasize the importance of maintaining consistency in the development teams to ensure software quality and minimize the likelihood of errors and vulnerabilities. This thesis presents practical strategies and solutions to address these challenges and enhance software development practices, with the ultimate goal of improving the overall quality of software products.

1.2 Purpose

The primary objective of this thesis is to provide development teams with strategies for improving software quality assurance, particularly when working on a long-term project. This can be accomplished by promoting a consistent code style, producing readable and maintainable code, minimizing bugs and errors, and reducing security vulnerabilities in the software. By implementing these strategies, development teams can avoid common pitfalls and improve software development practices, resulting in higher-quality software products that meet the needs of end-users and stakeholders alike. Ultimately, this thesis aims to assist development teams in enhancing their software development practices to achieve greater success and productivity in their work.

1.3 Study Structure

The study was conducted in a systematic manner, with each stage focusing on addressing a specific problem. The structured approach consisted of four sections:

The first section provides a comprehensive presentation of the problem, highlighting its key features and associated challenges. In the second section, a

detailed and specific methodology is presented, along with the special materials used to execute the solution. The third section presents the results obtained from the execution of the methodology, including brief descriptions, explanations, or instructions, and supporting visual aids such as tables, graphs and sample code or files. The discussion section follows, wherein the implications and significance of the obtained results were analysed and interpreted, leading to the formulation of conclusions and recommendations in the final section.

This structured approach ensured that each problem was addressed with the necessary rigor and provided a clear framework for presenting the findings, thereby enhancing the study's clarity and overall effectiveness.

2 Static Analysis

The absence of static code analysis in software development can lead to negative impacts, including an increased likelihood of introducing bugs and security vulnerabilities, reduced code quality, and decreased maintainability and scalability. To address this issue, developers can incorporate static code analysis tools into the development process. These tools identify potential issues early in the development cycle, making them easier and more cost-effective to address, and improve overall code quality, making the system more maintainable and scalable. This section emphasizes the importance of using static code analysis, discusses the negative impacts of not utilizing it, presents two widely used static code analysis tools (ESLint and Prettier) that can be employed to solve the problem, evaluates the outcomes of their implementation, and concludes with a discussion on the significance of incorporating static code analysis into the software development process.

Static analysis is a debugging technique employed to examine the source code of a software system while it is in a resting state (i.e., static) without the need to execute the program. This method is utilized to detect potential quality issues such as programming errors, coding standard violations, performance concerns, and security weaknesses. By utilizing static analysis, developers gain a comprehensive understanding of their codebase and can subsequently ensure compliance, safety, and security of the software system. [1]

Static analysis tools offer an automated solution to the process of detecting code quality issues and identifying flaws throughout the software development lifecycle. These tools are invaluable for monitoring the codebase and ensuring that it adheres to established coding standards and best practices. By providing real-time feedback on code quality and identifying potential vulnerabilities, static analysis tools can help developers address issues early in the development process, resulting in more efficient development practices and a higher-quality product.

Code analysis importance

Code analysis is a crucial element of the agile product development lifecycle, serving as a pre-emptive measure against potential issues, defects, errors, and bugs that may arise from the frequent changes to the source code that occur due to various factors such as evolving software needs and requirements driven by increasing customer demand, the need for code optimization resulting from inadequate code review processes during the software development cycle, and the ever-present threat of security breaches and vulnerabilities that require every layer of a software application to be fully fortified. By systematically analysing the codebase for issues related to broken code and bugs, code analysis provides developers with a comprehensive understanding of the software system, which enables them to take proactive measures to mitigate potential risks and ensure the highest level of code quality and security.

Code analysis techniques for optimal code quality

Code analysis is broadly categorized into two distinct types, namely static and dynamic analysis, each serving a specific purpose in detecting defects in software systems. While both approaches aim to identify potential issues, they differ in the stage of the development lifecycle where they locate defects. By combining static and dynamic analysis techniques, developers can proactively identify and address issues, resulting in software systems that are more efficient, secure, and resilient. The two types are described below.

1 Static Code Analysis

Static analysis is a process that scrutinizes the source code by debugging it and identifying potential issues, vulnerabilities, bugs, and security threats, without the need to execute the program. By examining the code at rest, static analysis provides developers with an understanding of the codebase, allowing them to ensure compliance with established coding standards and best practices. This

helps to identify defects in code quality early in the development process, leading to a strong and robust code base that is more efficient, secure, and resilient. [2]

2 Dynamic Code Analysis

Dynamic code analysis, also known as Dynamic Application Security Testing (DAST), is a powerful technique that identifies defects and vulnerabilities in both compile time and runtime by analysing the behaviour of the code after it has executed. DAST tools are executed on a running application, enabling them to detect a wide range of potential vulnerabilities that can occur at various stages of the application, such as preparing input data, running a test program, and analysing output data. By performing these critical steps based on the product's set parameters, DAST tools provide developers with a comprehensive view of their application's security posture, allowing them to identify and remediate potential security issues before they can be exploited by attackers. Consequently, dynamic code analysis is an essential tool in any comprehensive security testing program, helping developers to build secure and resilient software systems that meet the highest standards of quality and safety. [3]

Benefits of static code analysis

Static code analysis provides several benefits that can improve the efficiency, quality, and security of the software development process. First and foremost, automated testing using static code analysis tools saves significant time and manual effort by defining and configuring required test rules. Additionally, early detection and identification of issues and errors in the source code help prevent failures and reduce costs associated with fixing them later in the development lifecycle. Finally, static code analysis helps to improve source code quality and accuracy, ensuring that the code is compliant, secure, and robust against possible threats. [4]

2.1 Risks of Neglecting Static Code Analysis in Software Development

In the project, the developer team underwent constant changes over regular intervals of approximately four months. The short periods of time made the teams work under pressure to get tasks done on time before the deadline and meet coding and compliance standards. This, in turn, affected their performance during the application development phase and made them generally focus on the functionality and design aspects of the software without paying much attention to making the application robust enough to avoid security flaws. As a result, they were unconscious of the security risks caused by poorly designed and untestable code, which may have impacted the overall quality and security of the software.

Within the project, various types of defects have been identified, including variables with undefined values, declared but unused variables, violations of programming standards and syntax, inconsistencies in module and component interfaces, security vulnerabilities, and unused code. To address these weaknesses and improve the quality of the source code while mitigating risks and vulnerabilities as early as possible in the development lifecycle, the study was conducted to set up and execute static code analysis. By leveraging this automated approach, the study aimed to eliminate the need for time-consuming manual checks of security checklists and code reviews, while also enabling a more comprehensive and consistent analysis of the code. While the scope of the study was limited to static code analysis, it is an essential area of focus for software development teams looking to enhance the security, performance, and overall quality of the application.

2.2 Utilized Methods for Static Code Analysis

In order to address the issue, the decision was made to implement two static analysis tools, namely ESLint and Prettier, which have been deemed to be the most effective solutions for maintaining code quality and structure. By utilizing these tools in tandem, developers can enhance their confidence in the code, since Prettier is responsible for code formatting while ESLint is responsible for

ensuring code adheres to stylistic guidelines. Ultimately, this approach helps to improve overall code quality and promotes a more efficient and streamlined development process.

2.2.1 Selection Criteria for Best Tools

To ensure optimal results, it was imperative to carefully select a tool that could effectively perform static analysis of the source code, in a manner that best aligns with the project's requirements. Several key factors were taken into consideration during the selection process, including the programming language used in the project, which in this case is TypeScript.

Another key factor was the ease of integration with the development environment. To ensure a seamless integration process, it was important to select tools that offer ready-made plugins compatible with the development environment, in this case, Visual Studio Code. This would allow issues to be caught during the software writing process, enabling developers to address them promptly and efficiently.

Moreover, the chosen tools must provide clear and easily understandable documentation, explaining the configuration and analysis rules in detail. This would enable developers to utilize the tools effectively and resolve any issues that might arise during the analysis process. The analysis speed is also a crucial factor, as it can significantly impact the development process, particularly in large projects.

The type of reports generated by the tools is another important consideration. The chosen tools must provide comprehensive reports that highlight the number of errors sorted out by the level of certainty and detailed information about the type of each error. This level of detail is critical for developers to prioritize and address errors promptly and ensure that they do not negatively impact the application's overall quality.

Lastly, the licensing cost of the selected tools was taken into account. In this case, open-source libraries with free licenses were preferred. This not only saves costs, but also provides the added benefit of a vast community of users, who can contribute to the development and improvement of the tools. [5]

The selection process of a tool for static analysis of source code requires careful consideration of several key factors, especially when using a programming language like TypeScript. These factors include the programming language used in the project, ease of integration with the development environment, comprehensible documentation, analysis speed, comprehensive reporting, and licensing cost. The selection process requires balancing these factors to ensure optimal results.

2.2.2 ESlint

ESLint is a versatile linting tool for JavaScript that automatically detects inconsistencies and errors in ECMAScript/JavaScript code, enforcing a set of style, formatting, and coding standards across a codebase. As an open-source library, it is fully customizable, allowing developers to configure the desired rules to suit their coding practices and project requirements. With its extensive set of default rules and customizable nature, ESLint is an essential tool for developers seeking to write clean, reliable, and maintainable JavaScript code with confidence. By promoting code consistency and reducing the occurrence of bugs, ESLint helps developers to produce high-quality code that is easier to maintain and scale over time. [6]

How ESlint works

TypeScript ESLint Parser is a cutting-edge tool that generates an abstract syntax tree (AST) for TypeScript source code. This AST is then processed through a series of rules, each representing a set of restrictions that code blocks must follow to meet certain expectations. Detected issues typically take the form of syntactic,

semantic, or stylistic errors, and are reported in a detailed output that includes descriptions of each issue. Certain rules also offer automated correction features, enabling the tool to automatically correct issues, while other issues may need to be corrected manually. The result is a powerful and flexible tool that helps ensure TypeScript code conforms to best practices and delivers optimal results. [7] [8]

2.2.3 Prettier

Prettier is an advanced code formatting tool designed to enhance the readability and consistency of codebases. This opinionated tool supports a vast array of programming languages, including but not limited to JavaScript, JSX, Angular, Vue, Flow, TypeScript, CSS, JSON, GraphQL, Markdown, and YAML. By applying a consistent set of formatting rules, Prettier ensures that code is presented in an optimal way that is easy to read and understand, promoting best practices and enabling developers to focus on delivering high-quality code that is both efficient and effective. [9]

Enhancing Code Quality with Prettier

Prettier is a cutting-edge tool designed to streamline the development process by enforcing consistent code styling across entire codebases. This tool achieves this by parsing JavaScript code into an abstract syntax tree (AST) and replacing the original styling with pretty-printing of the parsed AST using a set of predefined rules. These rules consider a variety of factors, including line length and formatting standards, to ensure that code is presented in an optimal and consistent manner. By promoting consistency across the entire codebase, Prettier enables developers to work more efficiently and effectively, without the need to worry about formatting and styling issues. [10]

2.2.4 Custom Rules Configuration for Optimal Code Standards

ESLint is a highly flexible and customizable tool designed to meet the specific needs of different use cases. This powerful tool comes equipped with a wide range of built-in rules, which can be easily enabled or disabled to suit the project requirements. Developers can configure rules for an entire directory and its subdirectories either in a separate file, typically named as a .eslintrc.*, or bundled in an eslintConfig field within the package.json file. As illustrated in Code block 1, ESLint rules can be configured using the rules key along with an error or warning level and the desired options, making it simple and efficient to ensure that code adheres to the desired standards. [11]

```
"rules": {
    "react/react-in-jsx-scope": "off",
    "camelcase": "warn",
    "spaced-comment": ["warn", "always", { "markers": ["/"] }],
    "quotes": ["warn", "single", {"avoidEscape": true}],
    "no-multi-spaces": "warn",
    "no-duplicate-imports": "error",
    "no-debugger": "error"
},
```

Code block 1. Customizable rules in ESLint empower effortless alignment with desired coding standards and style.

Prettier is a versatile and widely used code formatter that offers an extensive range of formatting options across a multitude of programming languages. It is designed to be easily customizable through configuration files such as .prettierrc in JSON or YAML format, with the support of cosmiconfig. What sets Prettier apart from other code formatters is its intentional avoidance of global configuration to ensure that every team member receives consistent results. By default, Prettier's configuration is local, ensuring that the same consistent results are produced throughout the project. Code block 2 illustrates how easy it is to configure Prettier by simply modifying the options within the .prettierrc.json file. This flexible approach to code formatting allows developers to focus on writing

high-quality code while Prettier handles the styling, ensuring consistency throughout the codebase. [12]

```
{
  "semi": false,
  "tabWidth": 2,
  "printWidth": 100,
  "singleQuote": true,
  "trailingComma": "all",
  "jsxSingleQuote": true,
  "bracketSpacing": true
}
```

Code block 2. Configuring Prettier is effortlessly accomplished by modifying the options in the .prettierrc.json file.

2.3 Results

By adopting the appropriate tools, a fully automated workflow was established to effectively manage problematic coding patterns and noncompliant style guidelines, ultimately resulting in a more consistent and higher quality software product, while minimizing the associated effort and cost. The achieved results were highly satisfactory, affirming the success of the tool implementation in achieving the desired outcomes.

The initial analysis revealed that running ESLint and Prettier separately during static code analysis was not optimal, as these tools have overlapping rules that can potentially conflict with each other and lead to unexpected behavior. In order to ensure that each tool performs its intended role accurately and effectively, a specific strategy was implemented where Prettier handles formatting rules that govern code style, while ESLint focuses on improving code quality and detecting potential bugs. To accomplish this, rules that conflicted with Prettier were turned off in ESLint by using the eslint-config-prettier configuration and adding it to the extends array in the ESLint configuration file. Additionally, it was ensured that Prettier configuration was placed last, thus allowing it to override settings from other configurations. In summary, this approach ensures that Prettier is included

when calling ESLint from the command line. Code block 3 is a visual representation of how Prettier was appended to the ESLint file.

```
"extends": [
              "eslint:recommended",
              "plugin:react/recommended",
              "plugin:@typescript-eslint/recommended",
              "prettier"
],
```

Code block 3. Integrating Prettier with ESLint for improved code quality and consistency.

The comprehensive analysis report provided detailed information regarding the identified issues, including the number of problems, their type (i.e., error or warning), and their potential resolution (i.e., automatic or manual). Additionally, the report indicated the precise file location and line number where each problem occurred. This information proved to be instrumental in quickly identifying and addressing the issues, resulting in a more efficient and effective error resolution process.

ESLint is equipped to perform automatic code formatting for certain rule violations, such as adding a missing semicolon or removing multiple empty spaces or lines. However, for other rule violations that require fixing, an additional "--fix" argument can be used to format the written code in accordance with ESLint's rules. However, in the present study, this argument was not employed for reasons of software safety, particularly given the large size of the codebase. Running ESLint with "--fix" on existing code files could potentially introduce unintended errors and lead to file breakage. Therefore, each file had to be manually reviewed and handled individually to approve automatic repairs, which consumed considerable time that could have been better spent on other aspects of the development process. To mitigate this issue, code analysis could have been employed from the outset of the project.

Prettier provided a notable advantage through its convenient integration with VSCode, which allowed for automatic code formatting upon file change or save. This feature was particularly valuable as it obviated the need for developers to manually invoke Prettier, reducing the time required for code formatting and enhancing productivity. Additionally, the automatic formatting ensured that code was consistently formatted, making it easier to read and understand. By leveraging this feature, code could be written without worrying about adhering to formatting standards, resulting in a smoother development process overall.

2.4 Discussion

The study yielded promising results, as the employed static code analysis tools successfully detected various types of defects, with the majority being effectively addressed. However, certain difficulties and limitations stemming from design or methodology have impacted the programmatic, quantitative, and temporal aspects of the remediation process. It is worth noting that while static analysis tools are capable to detect potential defects within a function, but they are unable to verify whether the function is fulfilling its intended purpose. Additionally, it is important to acknowledge that certain security vulnerabilities, such as authentication issues and access control problems, may not be easily detectable through automated means. Furthermore, the presence of a significant number of unused files and excessive amounts of unreachable data hindered the speed of the process and potentially impacted the accuracy of the results.

2.5 Conclusion

Static code analysis is an indispensable tool in the software development process, facilitating the review of source code and enhancing its quality, thereby accelerating development processes. It ensures code uniformity, following specified rules to ensure consistency. Automated tools are faster than manual code reviews, pinpointing errors in the code and speeding up the process of defect detection and fixing. Difficult-to-read code can slow down teamwork, creating problems for new developers who may struggle to decipher the code.

Moreover, static code analysis can identify potential security vulnerabilities, helping to ensure that software is robust and resilient. By detecting issues early in the development process, static code analysis can save time and resources, avoiding costly errors and reducing the risk of bugs causing problems later on. In addition, static code analysis can help enforce coding standards and best practices, promoting good code maintenance and making it easier to maintain and update code over time. Finally, it can aid in the creation of more maintainable and sustainable code, helping to build software that can evolve and adapt to meet changing needs over time.

It is strongly recommended that developers perform static code analysis early on in the software development process. This approach can help identify potential issues before they compound into more significant problems, ultimately saving valuable time and resources in the long run. By detecting defects early, developers can avoid costly errors and minimize the time spent fixing issues later on in the development process.

Looking ahead, it is crucial for developers to remain mindful of the importance of static code analysis and to continue leveraging its benefits regularly. This practice can help ensure that the code remains consistent, clear, readable, and secure, even as it undergoes ongoing changes and updates. By maintaining a proactive approach to code quality and security, developers can help ensure that the software remains reliable and effective over time, meeting the evolving needs of end-users and stakeholders alike.

3 TypeScript

The importance of employing a type system in software development cannot be overstated. The benefits of TypeScript are numerous, ranging from improved code readability and maintainability to more efficient and reliable development processes. When TypeScript is omitted from a project, the repercussions can be severe.

Without TypeScript, codebases can quickly become disorganized and difficult to understand, especially as the project grows in size and complexity. This can lead to a higher likelihood of errors and bugs, which can be difficult and time-consuming to debug and fix. Additionally, omitting TypeScript can make it more challenging to collaborate effectively with other developers, as code changes become more difficult to track and understand.

To address the challenges of working without TypeScript, various methodologies and tools can be utilized. These might include more thorough testing, additional code review processes, and more stringent development standards. While these measures can help mitigate some of the risks associated with omitting TypeScript, they cannot fully replace the benefits of a robust type system.

In the following section, the specific methodologies and tools utilized to address the challenges of working without TypeScript in the project are introduced. A detailed analysis of the results obtained is provided and conclusions are drawn about the effectiveness of these approaches. Finally, the implications of the study and the significance of employing TypeScript in software development projects to ensure code quality, reliability, and efficiency are discussed.

TypeScript is a statically typed programming language that builds upon JavaScript's core features, adding an additional layer of static typing to aid in code reliability and maintainability. While JavaScript provides primitives like string and number, it lacks the ability to consistently enforce the types of these values. This is where TypeScript comes in, providing a robust type system to ensure code consistency and prevent common programming errors. By catching issues before runtime, TypeScript helps to reduce the risk of bugs and improve the overall quality of code. [13]

3.1 Impact of Ignoring Typing Layer on Software Quality

In JavaScript, the ability to reassign any variable to “any” type can result in challenges that are difficult to debug, especially in production environments. To

mitigate this issue, TypeScript offers a robust typing system that layers over JavaScript, providing enhanced type support and error prevention. TypeScript's type system allows developers to catch type errors during compilation, helping to reduce potential runtime issues. Moreover, the language ensures that the correct type is used when assigning variables, promoting efficient and reliable code. By leveraging TypeScript's built-in type checking and error-catching features, developers can streamline their development process and improve their code quality. The language's ability to provide better type support and error prevention makes it a preferred solution for software development projects that require reliability and scalability.

The project aimed to utilize TypeScript as a programming language, however, the developers frequently assigned the “any” or "unknown" type to most variables. As is customary, third-party functions and libraries were integrated into the codebase, leading to a situation where developers may not have had knowledge of the types returned by these functions. Consequently, to save time, they opted to use “any” as the variable type, thereby exposing the code to potential issues and bugs during runtime or code updates. By relying heavily on “any”, the type checking benefits offered by TypeScript were eliminated, effectively making the codebase more akin to JavaScript, and increasing the likelihood of bugs arising from faulty assumptions about variable types.

Defining variable types is an essential aspect of improving code quality and preventing critical issues during runtime. In this study, the goal was to explore the benefits of using TypeScript to detect type errors and enforce variable types, ultimately promoting clean and scalable code. By leveraging TypeScript, the study aimed to highlight unexpected behaviour in the program and reduce the likelihood of bugs. Throughout the study, various coding scenarios were examined, and the impact of using TypeScript on the overall code quality was considered. An improvement in code quality was noticed as TypeScript was used to detect type errors and enforce variable types, resulting in fewer bugs and issues being detected. It was found that implementing these practices allowed for consistency, reliability, and scalability in the code. In retrospect, the use of

TypeScript was demonstrated to be a valuable tool for promoting code quality and preventing critical issues during runtime.

3.2 Utilized Methods for TypeScript

The adopted tool in the project for both client-side and server-side execution was TypeScript, a free and open-source high-level programming language developed and maintained by Microsoft, it offers all of JavaScript's features, and an additional layer of TypeScript's type system. [14]

TypeScript is a strict syntactical superset of JavaScript designed for detecting errors in code without running it, so it checks a program for errors before execution, and does so based on the kinds of values. It does not consider any JavaScript code to be an error because of its syntax, just it adds rules about how different kinds of values can be used. [15]

As a principle, TypeScript preserves the runtime behaviour of JavaScript code and never changes it, which means the transition between the two languages is done easily without worrying about subtle differences that might stop program working. [16]

TypeScript never changes the behaviour of the program based on the types it inferred, so once the code is compiled, the resulting plain JS code has no type information, because when TypeScript's compiler is done with checking the code, it erases the types to produce the resulting "compiled" code. [17]

3.2.1 TypeScript Compilation

Since browsers cannot directly execute TypeScript code, it must first be converted into JavaScript code through a process called "transpiling". This is done using the TypeScript compiler, known as tsc. To use the tsc compiler, developers must first configure it by creating a tsconfig.json file. This file specifies the root files of the project, as well as the compiler options required to compile

the code. The directory where the tsconfig.json file is located is considered the root of the project. By using the tsc compiler and configuring the tsconfig.json file, developers can ensure that their TypeScript code is properly transpiled into JavaScript code that can be executed by browsers. Understanding this process is crucial for building and deploying TypeScript projects, as it ensures that the code will function as expected in the target environment. [18]

3.2.2 Structure of tsconfig.json

The tsconfig.json file plays a crucial role in configuring the TypeScript compiler to accurately transpile TypeScript code into JavaScript. It is structured as a JSON object, containing properties and values that define both the root options and compiler options. Understanding the distinction between these two types of options is essential for properly configuring a TypeScript project. The root options are used to configure the overall behaviour of the TypeScript compiler and the project structure, as well as how the output JavaScript should be generated, while the compiler options configure the behaviour of the compiler itself to ensure accurate and efficient compilation of TypeScript code. By accurately configuring these options in the tsconfig.json file, the TypeScript compiler can effectively transpile TypeScript code into JavaScript code that can be executed in web browsers. [19]

The tsconfig.json file can be customized to specify various properties and values, here are some of the key properties that can be included in the tsconfig.json file to allow the configuration of a TypeScript project to be tailored to specific needs:

- "compilerOptions" is a configuration object in TypeScript that contains a set of options for the TypeScript compiler, such as the target version of JavaScript to compile to, the module format to use, and the level of strictness for type checking. These options make up the bulk of TypeScript's configuration and cover how the language should work.

  Some common options that can be specified in the "compilerOptions" object include:

  "target": Specifies the version of ECMAScript that the TypeScript code should be compiled to. For example, "ES5", "ES6", "ESNext".

"module": Specifies the module format of the compiled JavaScript code. For example, "commonjs", "amd", "es2015", "esNext".

"strict": Enables strict type checking and other strictness-related options in TypeScript.

"sourceMap": Generates a source map file that maps the compiled JavaScript code back to the original TypeScript source code.

"outDir": Specifies the directory where compiled JavaScript files should be outputted.

"declaration": Generates TypeScript declaration files (.d.ts) for TypeScript code, which provide type information for external consumers of the TypeScript code.

The project was divided into two separate repositories on GitHub - one for the front-end and one for the backend. As a result, two independent TypeScript configuration files had to be created, one for each repository.

Code block 4 provides an example of how the "compilerOptions" configuration was defined within a "tsconfig.json" file for the backend of the project. This configuration specifies that the code should be compiled to ECMAScript 6 and utilize the CommonJS module format. The resulting JavaScript files should be stored in the "./build/" directory. In addition to these fundamental options, the strict settings have been enabled to enforce additional checks and constraints on the code, which can enhance the code's overall quality. Other options have also been enabled, such as the ability to import JSON files and to facilitate the use of older-style module imports. By utilizing a "tsconfig.json" file in this way, fine-grained control can be exerted over the TypeScript compilation process, and it ensures that the resulting JavaScript code meets the specific requirements of the project. This allows the TypeScript environment to be tailored to the project's needs, providing developers with the ability to customize the compilation process according to the project's unique requirements.

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "./build/",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "resolveJsonModule": true
  }
}
```

Code block 4. Customizing TypeScript environment in the backend with strict settings and other options using 'compilerOptions' for fine-grained control over code quality.

In Code block 5, the "compilerOptions" configuration of the project is demonstrated through a "tsconfig.json" file for the frontend. The configuration encompasses various options such as the target ECMAScript version to compile the code to, the libraries to include, and whether or not to check against them.

Additional options include enabling synthetic default imports, enforcing consistent file naming conventions, specifying the module format, and setting the module resolution method to the Node.js module system. The configuration also enables the use of JSON files as modules and isolated modules. It's worth noting that the "noEmit" option is set to true, indicating that no JavaScript files should be generated as output, and the configuration file is intended only for checking and validating purposes. In this way, the configuration can be set up to ensure that the desired output is produced by the TypeScript compiler and that the resulting JavaScript code meets high-quality standards, in accordance with the project's specific requirements.

```
{
  "compilerOptions": {
    "target": "es6",
    "lib": ["dom", "dom.iterable", "esnext"],
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react",
    "downlevelIteration": true,
    "allowJs": true
  }
}
```

Code block 5. Fine-tuning the TypeScript environment for the frontend with 'compilerOptions' in the tsconfig.json file to enhance code quality and meet specific requirements.

- "files": This property specifies an array of individual TypeScript files that should be included in the compilation process. If this property is used, the compiler will only compile the files listed in this array, ignoring all others.

The "files" property can be particularly useful in larger projects, where it can help to avoid compiling unnecessary files and reduce the build time. By carefully selecting which files to include, developers can optimize the compilation process and ensure that only the necessary files are compiled.

In Code block 6, the inclusion of specific TypeScript files in the compilation process is determined by the "files" property in the tsconfig.json file. The array contains seven files: "allUsersActions.ts", "breadcrumbActions.ts", "types.ts", "formActions.ts", "parser.ts", "utilities.ts", and "tsc.ts", and any other TypeScript files in the project are excluded. The "files" property offers a more granular approach to file inclusion, enabling developers to precisely control which files are included in the compilation process. In contrast, the "include" and "exclude"

properties provide broader control by specifying a pattern of files to be included or excluded from the compilation process. It is worth noting that the "files" property supersedes the "include" and "exclude" properties, meaning that any TypeScript files specified in the "files" property will override any patterns specified in the "include" and "exclude" properties.

```
{
  "files": [
    "allUsersActions.ts",
    "breadcrumbActions.ts",
    "types.ts",
    "formActions.ts",
    "parser.ts",
    "utilities.ts",
    "tsc.ts"
  ]
}
```

Code block 6. Precising Control over TypeScript Compilation with 'files' Property in tsconfig.json.

- "extends":This property enables the extension of the configuration from another tsconfig.json file, making it possible to share common settings among multiple projects. This feature proves to be valuable in scenarios where consistency is required across multiple TypeScript projects.

Code block 7 illustrates the use of the "extends" property in the tsconfig.json file. The "extends" property allows a TypeScript configuration file to extend or inherit settings from another configuration file. In this example, the "./tsconfig" file is the configuration file from which the current configuration file inherits settings. This is particularly valuable in projects where multiple applications share similar configurations, as it reduces duplication and promotes consistency. An excellent use case for the "extends" property is in scenarios where a development team is working on multiple applications that require a common set of dependencies or build requirements. By defining these settings in a single configuration file and then extending them in other files, the team can ensure that all their projects utilize the same set of libraries and tools, which can improve collaboration and productivity.

```
{
  "extends": "./tsconfig"
}
```

Code block 7. Streamlining TypeScript configuration management with 'extends' in tsconfig.json enables the inheritance of settings from a common configuration file and reduces duplication across multiple projects.

- "include": In the tsconfig.json file, the "include" property is used to specify an array of file globs or patterns that are to be included in the compilation process. These can include individual files, directories, or a combination of both. This property is particularly useful in large projects that contain a multitude of source files spread throughout a nested directory structure. By including specific files or directories in the compilation process, developers can ensure that only the necessary TypeScript code is transpiled into JavaScript.

In the tsconfig.json file example, Code block 8 provides a practical illustration of how the "include" property can be used. The example shows that by specifying the "src" and "tests" directories within the "include" property and setting the "recursive" option to true, TypeScript will compile all files in these directories, along with their respective subdirectories. This can be particularly useful when working on large projects with multiple files and directories, as it simplifies the TypeScript compilation process.

```
{
    "include": [
        "src/**/*",
        "tests/**/*"
    ]
}
```

Code block 8. Using the "include" property in tsconfig.json to include files from specific directories and their subdirectories in the TypeScript compilation process.

- "exclude": This property specifies an array of file globs or patterns that should be excluded from the compilation process. This can be useful for excluding files that should not be compiled, such as documentation files, configuration files, and build output files.

Excluding these files can help to improve the efficiency of the compilation process and reduce the overall size of the compiled output.

In the given example, Code block 9 illustrates that the exclude property is set to ./**/*/ignore.ts, which means any file with the name "ignore.ts" located in any directory and any subdirectory of the project will be excluded from the compilation process. This can be useful when you have files that should not be compiled or when you want to ignore specific parts of your project for various reasons. This can be useful when files should not be compiled, or specific parts of a project need to be ignored for various reasons.

```
{
    "exclude": [
        "./**/*/ignore.ts"
    ]
}
```

Code block 9. Excluding files from the TypeScript compilation process can be easily done using the "exclude" property in the tsconfig.json file.

- "references": This property is utilized to reference other TypeScript projects that the current project relies on, and it helps to manage dependencies between different projects by ensuring that they are built in the correct order. This feature was introduced in TypeScript 3.0 to allow programs to be structured into smaller pieces for faster build times, enforce logical separation between components, and organize code more effectively.

Code block 10 illustrates the usage of the "references" property, where a reference to another TypeScript project is made. In the given example, the references property is used to reference another TypeScript project located in a directory that is one level up from the current project directory. The "path" property is set to "../src", which means that the current project depends on the TypeScript project located in the "../src" directory.

This can help to manage dependencies between different projects and ensure that they are built in the correct order. By referencing other projects, the TypeScript compiler can build the dependencies first before building the current

project, which can help to speed up build times and ensure that the code is organized in a logical and consistent way.

```
{
    "references": [
        { "path": "../src" }
    ]
}
```

Code block 10. The 'references' property is used to reference another TypeScript project, allowing for better dependency management and optimized build times.

The tsconfig.json file plays a fundamental role in configuring the TypeScript compiler, facilitating efficient transpilation of TypeScript code into high-quality JavaScript that aligns with project specifications. Accurate configuration of this file is crucial for achieving precise and efficient compilation, allowing for greater control and accuracy in the TypeScript-to-JavaScript conversion process. With proper configuration, the tsconfig.json file can help ensure that the resulting code is of the highest quality and meets all project requirements, emphasizing the importance of its accurate and thorough setup.

3.3 Results

Throughout the study, the TypeScript Compiler tool was utilized to perform rigorous type checking on the program. By analysing the code at compile-time, the tool was able to ensure that variables and functions were used in a manner consistent with their intended types. Type-related errors were able to be identified and resolved before the program was ever run, allowing for more efficient debugging and a more stable final product. Overall, the use of the TypeScript Compiler was instrumental in guaranteeing that the program computed with the expected values, and helped to produce clean, well-documented code that could be more easily maintained and expanded upon in the future.

TypeScript, as a robust type system, proved highly effective in detecting and preventing numerous type errors and successfully brought attention to potential

bugs that could be prevented during the coding process by enforcing variables and other data structures to be declared with specific types to determine what values the variables could have and what operations could be performed on them. Figure 1 demonstrates how type error were identified and highlighted when “any” was used as a type specifier.

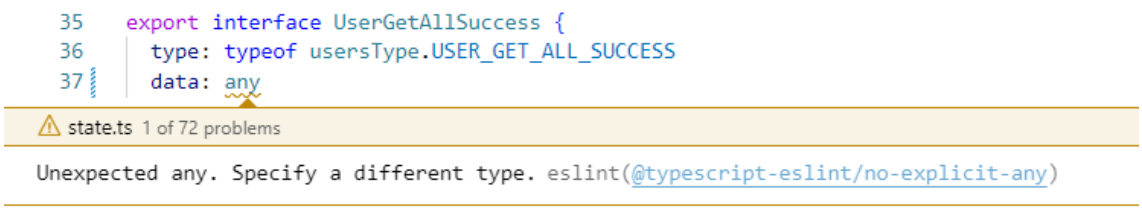

Figure 1. The effectiveness of TypeScript's type system in detecting and preventing type errors, where the use of “any” as a type specifier is highlighted as a type error.

Figure 2 illustrates how the previously detected type error has been resolved by replacing the generic “any” type with an explicit type declaration. The new type declaration specifies that the data in question is an array of objects, with each object conforming to the User data type.

This is a significant improvement over using “any” as the data type, as it removes any ambiguity about the structure of the data and allows for more precise type checking. By explicitly specifying the data type, TypeScript is able to enforce stricter typing rules, which helps catch errors earlier in the development process.

By replacing “any” with an explicit type declaration, developers can reduce the likelihood of errors occurring in their code and make it easier to maintain and modify over time. This is particularly important in larger codebases, where a single mistake can have far-reaching consequences.

Overall, the use of explicit type declarations is a best practice in TypeScript development and is essential for building robust and reliable software easy to maintain, and less prone to errors.

```
35  export interface UserGetAllSuccess {
36    type: typeof usersType.USER_GET_ALL_SUCCESS
37    data: User[]
38  }
```

Figure 2. Replacing “any” with an explicit type declaration in TypeScript will improve code reliability and make it easier to maintain.

The application of static typing in TypeScript proved to be a highly effective technique, as it had a profound impact on the quality, efficiency, and reliability of the software. By enforcing strict type checking, TypeScript was able to detect potential bugs in code before it was executed, minimizing the likelihood of errors and enhancing the stability of the software. In addition to increasing reliability, static typing also led to enhanced run-time efficiency. Since the TypeScript compiler checked for type errors before execution, there was less need for type checks during runtime, which reduced overhead and resulted in faster, more efficient code. Another benefit of static typing was that it encouraged developers to adopt a more structured and organized approach to coding. By requiring explicit declaration of variable and function types, TypeScript promoted code readability, understandability, and maintainability over time. This helped reduce the likelihood of bugs and errors that can arise from complex, poorly structured code. Moreover, the use of static typing sped up the development process by enabling developers to catch and fix errors early in the coding cycle. This saved valuable time and resources, allowing developers to focus on adding new features and functionality to the software.

The study succeeded in using TypeScript's inbuilt support for interfaces, which is an important typing feature, to define the specifications of an entity. This allowed the study to catch related errors using TypeScript's type inference capabilities, which examine the name of an object's property and its associated data type. TypeScript interfaces provide a way to define the structure of an object, specifying the properties and their types. This ensures that objects are created consistently throughout the codebase, which helps to prevent errors and maintain code quality. The use of interfaces also promotes code readability and maintainability by providing a clear definition of the properties that an object should have. By

using TypeScript's type inference capabilities to examine the name of an object's property and its associated data type, the study was able to detect errors early in the development process. This helped to ensure that errors were caught and fixed before they caused any serious issues in the software. Code block 11 was used in the study to illustrate an example of an interface structure, which contains the names of all the properties of a UserInformation object along with their associated data types. This interface structure provided clear guidelines for developers to follow, ensuring that the UserInformation object was created consistently throughout the development process.

```
export interface UserInformation {
  name: string
  email: string
  city?: string
  street?: string
  zipCode?: string
  phoneNumber?: string
  website?: string
  licenses?: string
  profilePicture?: string
  category?: string
}
```

Code block 11. Using TypeScript interfaces to define object specifications and leveraging its type inference capabilities helps to catch related errors early in development. and improve code quality.

The study successfully leveraged TypeScript's support for enums, a powerful feature that provides a way to define a set of named constants with specific values. Unlike regular JavaScript, TypeScript provides both numeric and string-based enums. [20]

This feature enabled the study to define a variable as a set of predefined constants, ensuring that it can only be equal to one of the predefined values. This allowed the study to clearly document intent and create a set of distinct cases that could be easily referenced and understood throughout the code. One of the major benefits of using enums was the improvement in code readability and maintainability. By providing a clear definition of the values that a variable can

take, the study was able to improve the structure and organization of the code. Additionally, by using the standard structure for string-based enums, the study was able to define a set of distinct cases that were easy to reference and document. Code block 12 provided a visual representation of the standard structure for string-based enums, which includes a list of constant values, each with a unique string identifier. This structure was used in the study to define a set of distinct cases, making it easier to document the intent of the code and ensure consistency throughout the project.

```
export enum usersType {
  USER_CREATED_REQUEST = 'USER_CREATED_REQUEST',
  USER_CREATED_SUCCESS = 'USER_CREATED_SUCCESS',
  USER_ACTION_FAILURE = 'USER_ACTION_FAILURE',
  USER_GET_ALL_REQUEST = 'USER_GET_ALL_REQUEST',
  USER_GET_ALL_SUCCESS = 'USER_GET_ALL_SUCCESS',
  USER_GET_CURRENT_REQUEST = 'USER_GET_CURRENT_REQUEST',
  USER_GET_CURRENT_SUCCESS = 'USER_GET_CURRENT_SUCCESS',
  USER_UPDATE_REQUEST = 'USER_UPDATE_REQUEST',
  USER_UPDATE_SUCCESS = 'USER_UPDATE_SUCCESS',
  USER_DELETED_REQUEST = 'USER_DELETED_REQUEST',
  USER_DELETED_SUCCESS = 'USER_DELETED_SUCCESS',
  USER_UPDATE_STATUS_REQUEST = 'USER_UPDATE_STATUS_REQUEST',
  USER_UPDATE_STATUS_SUCCESS = 'USER_UPDATE_STATUS_SUCCESS',
  USERSTATUS_UPDATE_REQUEST = 'USERSTATUS_UPDATE_REQUEST',
  USERSTATUS_UPDATE_SUCCESS = 'USERSTATUS_UPDATE_SUCCESS',
}
```

Code block 12. The standard structure for string-based enums, which allows for a set of distinct cases to be easily referenced and understood throughout the code.

The study demonstrated the benefits of using TypeScript, a robust type system that enforced strict type checking and promoted code organization, readability, and maintainability. By analysing code at compile-time, TypeScript's inbuilt tools caught potential bugs early, leading to more stable and efficient software. Features such as interfaces and Enums provided clear guidelines for developers, improved code structure and readability, and reduced the likelihood of errors. In summary, the study results indicate that incorporating TypeScript's static typing

into the software development process significantly improves software quality, efficiency, and reliability.

3.4 Discussion

TypeScript's strong static typing feature has been widely adopted by software developers for its ability to catch type-related errors early in the development process. The study was able to leverage from this feature, which resulted in more reliable and efficient code. By requiring developers to declare variable types, TypeScript prevents common errors that can lead to bugs and poor performance in production. This means that the codebase is more robust and maintainable, and the development process is streamlined, leading to faster debugging and refactoring.

TypeScript is a strongly typed language, which means that every variable in a TypeScript program must have a specific type. This ensures code correctness but can also be time-consuming for developers to manually declare all the types for their variables. However, the use of IDE like Visual Studio Code was a significant advantage for the study, enabling developers to work more efficiently and effectively as it comes with built-in TypeScript intelligence that can automatically infer variable types based on the code context. This can save developers time and effort when writing code. Additionally, when the variable types are explicitly declared, the IDE can provide accurate suggestions, code navigation, and autocompletion, making it easier for developers to write correct code and navigate through the codebase. Furthermore, Visual Studio Code can also perform real-time error checking, highlighting any syntax errors, semantic errors, or type mismatches as the developer writes code. This can help catch errors early in the development process, leading to more efficient debugging and reducing the risk of introducing bugs into the codebase. Finally, when errors are detected, VS Code can help with debugging by providing detailed information about the error, including the line number and the specific error message.

TypeScript's compile-time error detection is a powerful feature that allows developers to catch type-related errors before running the program, which can save significant time and effort during the debugging and refactoring processes. This feature was utilized in the study, resulting in more efficient development processes. When developers declare the type of variables in TypeScript, the compiler can catch type errors immediately and highlight them as they occur. This means that developers can identify and resolve issues early in the development cycle, resulting in code that is more dependable and optimized for performance. By providing early feedback on type-related errors, TypeScript can significantly improve the software development process by reducing the likelihood of errors in production. It achieves this by enforcing strong static typing, which catches potential bugs early before moving to production phase. Applying TypeScript to the project also gives the codebase more structure, enhances its readability and keeps it robust, organized, and maintainable.

Applying TypeScript to the project added a layer of structure and organization by requiring developers to declare variable types and adhere to strict syntax rules. This resulted in more readable and maintainable code, as well as easier collaboration between team members. Additionally, TypeScript's ability to detect and catch errors at the compile stage and prevent bugs from slipping through to production leads to a more robust and stable codebase. This allows for smoother maintenance and reduces the likelihood of costly errors or downtime. Overall, the use of TypeScript can greatly enhance the quality and reliability of a project's codebase.

The study greatly benefited from the powerful object-oriented programming features provided by TypeScript, including the ability to define interfaces and classes. An interface is a way to define the structure and contract that a class should follow. It specifies the method signatures and property types that a class should have but does not provide the implementation. Instead, the class that implements the interface defines all its members. The primary purpose of an interface was to provide a level of abstraction that allowed developers to separate the concerns of different parts of their code. This helped make code more

modular, reusable, and easier to maintain over time. In a research study context, this led to more efficient and effective development, as it allowed researchers to more easily isolate and test different parts of the codebase. In TypeScript, interfaces were only used at development time and were removed by the compiler when it generated the JavaScript code. This meant that they had no impact on the runtime performance of an application but could provide significant benefits during the development process. By using interfaces, developers could create well-organized and highly structured codebases that were easier to understand, modify, and extend. This was particularly useful in the context of a research study, where there may have been a need to modify and extend the codebase over time as new findings emerged or new research questions were posed.

3.5 Conclusion

Statically-typed programming languages, such as Java and C++, were developed with stricter typing rules enforced at compile-time to detect and prevent type errors at an early stage of the development process. This leads to the production of clear and expressive code. Conversely, dynamic, and weakly-typed languages, such as JavaScript, perform type checking at runtime based on contextual information and data, which allows variables to be assigned different types after initialization, as typing is associated with the variable's value rather than the variable itself.

With the additional strong typing capabilities in TypeScript, developers can benefit from stricter rules for defining data types, which enable them to detect type errors at compile-time before running the program. This results in considerable time and effort savings, as compile-time errors can be easily addressed during the code development phase, leading to the creation of cleaner and better-documented code.

The project was initially intended to use TypeScript as its primary programming language, which is known for its static typing capabilities that can help catch type errors early in the development process. However, upon reviewing the written

code, it was discovered that the code was written without the proper use of types, with the type “any” being used extensively throughout the codebase.

The use of “any” type means that the code was written without explicitly defining the data types for variables, functions, and other components, making it more difficult to ensure code correctness and maintainability. This lack of proper type annotations can lead to more bugs and errors during the running time of the program, as the TypeScript compiler is forced to disable type checking for variables and returned values of call-backs and ignore the comments around them. This can ultimately lead to a more challenging debugging process, as it can be difficult to track down and fix errors that arise from incorrect data types. To mitigate these issues and ensure the long-term maintainability and stability of the codebase, it is important to use TypeScript's strong typing capabilities effectively and consistently throughout the development process. To address this issue, the study aimed to revisit the codebase and update it to use TypeScript's strong typing capabilities more effectively, starting by creating interfaces to define the structure and contracts that classes and functions should follow, to make it easier to catch errors related to incorrect usage of objects and functions. Additionally, adding explicit type annotations to variables and function parameters, to clear what types are expected and reducing the risk of introducing errors.

Throughout the study, the effective utilization of TypeScript resulted in the successful detection and resolution of type errors in numerous files located throughout the project's various folders. This was accomplished by capitalizing on the inherent strong typing capabilities that TypeScript provides. However, despite the progress made, there are still number of files where type errors had not been handled. As a result, it is highly recommended that future work continues to use TypeScript and focuses on resolving these remaining issues. It may also be beneficial to evaluate and potentially delete any unreachable files, as these could introduce unnecessary complexity and maintenance overhead. By continuing to use TypeScript and prioritizing type safety, the project can ensure a more stable and reliable codebase over time.

4 Unit Testing and Integration Testing

The purpose of this section is to address the issue of missing code tests encountered during the project's development. The absence of code testing had a significant impact on the development process, leading to delays, errors, and a decrease in overall software quality. Thus, this section aims to highlight the importance of code testing in ensuring software quality, identifying and resolving bugs early in the development cycle, and meeting user requirements.

This section presents solutions for creating a suitable environment with the necessary assets for running the tests, such as tools, frameworks, and automated testing procedures. By implementing these solutions, developers can ensure that their software meets the required standards of quality and functionality, resulting in better user satisfaction and trust.

Overall, this section emphasizes the significance of code testing in software development and the detrimental effects of neglecting it. The solutions presented in this section will serve as a roadmap for developers to establish an effective code testing process, leading to a higher quality and more reliable software product.

The software testing process typically involves dividing testing into four primary levels, including unit, integration, system, and acceptance testing. This study, however, focuses primarily on API integration testing, which is a critical part of software development. API integration testing is a type of integration testing that focuses on testing the interactions between different software components, including servers, databases, and applications. The study provides an overview of the software testing process and the different testing levels, with a particular emphasis on API integration testing. The study also explores the significance of API integration testing, its challenges, and the best practices for designing and executing effective API integration tests. By focusing on this specific testing level, this study aims to provide insights that can help developers improve the quality and reliability of their software products.

4.1 Consequences of Untested Code in Software Development

The software development in this project was missing an essential phase, which is unit and integration testing. Writing code without testing is a high risk, because it relies on the skills and the knowledge of the programmer and as codes are written manually it is so possible to make mistakes, which can cause trivial or catastrophic errors, defects, or failure at any stage of the software development life cycle, also making changes and submitting potentially non-working code to a common branch makes the processes of development and code refactoring to be complicated.

Since many programmers have worked on this project, lines of codebase has been always piling up in the version control system for a build without a steady review or testing, which negatively affected the development process time to become much longer, that the developers are always required to work for long time to find out the sources and causes of the code breakdown and to figure out the appropriate solutions to fix them to be able to resume the development process. As the entire code has been written without tests, so it is unreliable, and it has been difficult to identify the root of many detected problems and issues in the codebase and understand where bugs come from and when they occur.

The best practice to find out if a feature is likely to cause a production downtime is to test that feature, therefore the study aimed to set up an effective environment that provides the best approaches to the testing process to implement early testing to shorten error logs by helping the developers to benefit from writing testable code to prevent the unnecessary effort of searching for solutions to the discovered defects and bugs before the delivery to the client, reduce code complexity and eliminate the unnecessary parts, make the code more robust and easier to maintain, help developers to detect problems and find bugs easily with less effort and fewer resources, guarantee the quality and high-performance of the software and make it more reliable and easier to use.

The critical role of code testing in software development

Code testing is an essential part of software development as it ensures the quality and reliability of software products by detecting and eliminating bugs and errors. It involves running various tests on the code to identify any issues that could affect the program's performance or stability. Testing allows developers to identify and fix bugs and errors in the code before they reach end-users, reducing the risk of product failures, security breaches, and other issues. Code testing also helps ensure that software meets the requirements and specifications set out by clients or stakeholders, ensuring customer satisfaction and trust.

One of the most significant benefits of code testing is that it helps identify issues early in the development process, making it easier and less expensive to fix. By catching problems early, developers can avoid the need for more extensive and costly fixes down the line, which can save time, money, and resources in the long run. Additionally, code testing helps identify areas of the code that may need improvement, leading to better overall code quality and maintainability.

Exploring the Levels of Code Testing

There are several levels of code testing in software development, beginning with unit testing at the individual component level, followed by integration testing to ensure proper interactions between components, system testing to validate the entire system, and acceptance testing to confirm that the software meets the specified requirements and is ready for release. These levels of testing work together to provide thorough coverage, enabling the delivery of a stable, reliable, and high-quality software solution.

- Unit testing

A unit testing is the first level of functional testing that verifies the functionality of a single, isolated component of a larger system. This component is typically a function, method, or class within the codebase, and it is tested in isolation from

any other components that may depend on it. The purpose of this isolation is to ensure that any failures or issues that arise during the test can be easily traced back to the specific component being tested. Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing.

Unit tests are designed to be automated, meaning that they are written as code and can be executed automatically by a testing framework. They are typically written by developers as part of the software development process, and they are run frequently during the development cycle to catch bugs and issues early on. This helps to ensure that the code is stable and reliable before it is released to users. [21]

Unit test provides several valuable advantages, including the early detection of bugs, which reduces defects in newly developed features and when changing existing functionality. Catching bugs early in the development process means they can be addressed before they propagate to higher levels of the system, which ultimately leads to a more stable and reliable software product. Additionally, unit testing helps to reduce the total cost of testing by detecting defects and bugs in the early development phase before they become more complex and expensive to fix. This approach also simplifies the refactoring of code and improves design, making it easier for developers to make changes to the codebase without introducing new issues. Moreover, when integrated with the build process, unit tests can improve the quality of the build, ensuring that the software is free from defects and behaves as expected. Finally, unit testing encourages developers to write modular, well-structured code, which is easier to maintain, reduces technical debt, and improves overall software quality. [22]

- Integration testing

Integration testing is a crucial stage in the software testing process that occurs after unit testing. It involves testing individual components or units of the software in a group, with the objective of identifying defects at the point of interaction between the integrated components or units of an application before they are released to production.

The primary goal of integration testing is to ensure that the different modules, which may have been developed by different programmers, work correctly as intended when combined and that data can be correctly passed between them. Integration test checks the correctness of communication, performance, and reliability among all the modules in the system.

The integration testing process can involve several approaches, such as top-down testing, bottom-up testing, and a combination of both approaches. It can be conducted manually or automated, depending on the complexity of the software being tested. [23]

Integration test provides several benefits that help improve the quality, reliability, and performance of the system. One of the key advantages of integration testing is that it guarantees the performance of integrated modules, even when designed by different software developers. This is particularly important in complex systems that rely on multiple modules working together to function correctly. Another significant advantage of integration testing is that it can identify integration issues early in the development process. Finding and resolving issues at this stage is generally easier and less expensive than finding them later, saving developers time and effort. Additionally, integration testing improves test coverage and reliability by testing the interactions between modules.

By catching defects early, integration testing can reduce the overall cost of maintaining a system over its lifetime. This is because it is generally less expensive to fix issues during development than after deployment. Finally, integration testing can help accelerate the development process, reducing time-to-market and giving developers a competitive edge in the marketplace. [24]

- System testing

System testing is a type of software testing that evaluates the entire system or software application, rather than its individual components or modules. It is typically conducted after integration testing and before acceptance testing to

ensure that the system meets the specified requirements and works as expected in a real-world environment. [25]

System testing involves testing the system's functionality, performance, reliability, and security by simulating real-world scenarios and inputs. This includes testing the system's user interfaces, APIs, hardware, software, and network connectivity, as well as testing for compatibility with different operating systems, devices, and browsers.

The goal of system testing is to identify any defects or issues that may have been missed during earlier testing phases and to ensure that the system is ready for deployment. This is typically the final testing phase before the system is released to users or customers. System testing is an essential aspect of the software development process, as it helps ensure that the system meets the needs of its users and functions as expected in a real-world environment.

- Acceptance testing

Acceptance testing is a vital phase of software testing that validates whether a software application meets the business and user requirements and is ready for deployment. It is typically the final phase of testing, performed by end-users or customer representatives, and tests the application's functionality and usability using real-world scenarios. There are two main types of acceptance testing: User Acceptance Testing (UAT) and Operational Acceptance Testing (OAT), each with a specific focus on validating the functionality and ability of the software to perform in its intended environment. UAT verifies that the software meets the needs of its intended users and is acceptable for deployment, while OAT evaluates the software's ability to perform in its operational environment. [26]

4.2 Utilized Methods for Testing Process

The application has been developed by TypeScript language, so Jest is a great option for implementing unit and integration testing for TypeScript projects due to its wide benefits.

Jest is a widely used, open-source JavaScript testing framework that has gained popularity in recent years for its ability to perform both unit and integration testing. Built by Meta (Facebook), Jest's design and features make it a great choice for testing TypeScript projects, providing many benefits to developers.

One of the main advantages of Jest for TypeScript projects is its built-in support for TypeScript. TypeScript is a typed superset of JavaScript that allows developers to write more reliable and maintainable code. However, testing TypeScript code can be challenging, especially when dealing with type checking and interfaces. Jest simplifies this process by providing built-in support for TypeScript, making it easy to write tests for TypeScript projects without requiring any additional configuration or setup. [27]

In addition to its support for TypeScript, Jest is known for its speed and reliability. Jest uses a highly optimized test runner that runs tests in parallel, making it faster than other testing frameworks. Jest also has built-in features for snapshot testing, which makes it easier to test complex components and ensure that they are working as expected. [28]

Another benefit of Jest is its ease of use. Jest has a simple and intuitive API that makes it easy to write tests. It also has built-in matchers that allow developers to write more expressive and readable tests. Jest's test runner provides instant feedback and displays test results in an easy-to-read format, making it easy for developers to identify and troubleshoot any issues that arise. [29]

Jest also has comprehensive documentation that covers all aspects of testing, including unit and integration testing. This documentation makes it easy for developers to get started with testing and to troubleshoot any issues that they may encounter. Jest also provides many plugins and extensions that can be used to extend its functionality, providing even more flexibility for developers. [30]

Finally, Jest has a large and active community of developers who contribute to the framework, provide support, and share their knowledge and expertise. This

community ensures that Jest is continuously improved and updated to meet the needs of developers.

the best option to implement unit and integration testing as it is the most popular framework for testing TypeScript and react components.

Jest is a delightful JavaScript testing framework with a focus on simplicity. It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more. It is open source designed by Facebook to ensure correctness of any JavaScript codebase. It allows developers to write tests with an approachable, familiar, and feature-rich API that gives results quickly. Jest is well-documented, requires little configuration and can be extended to match the desired requirements. [31]

To make Jest work with TypeScript, Jest transformer had to be used. Ts-jest is a Jest transformer with source map support that lets developers use Jest to test projects written in TypeScript. It supports all features of TypeScript including type-checking. [32]

4.3 Results

The study successfully achieved integration tests for the authentication API, as demonstrated by the results in Figure 3. These tests not only verified that individual components of the authentication system were functioning correctly, but also tested the system as a whole in a more realistic environment. By running these integration tests, the study was able to identify and address issues that could only be detected at the system level, rather than in individual unit tests. Overall, the successful integration testing of the authentication API was a critical step in ensuring the reliability and functionality of the software system.

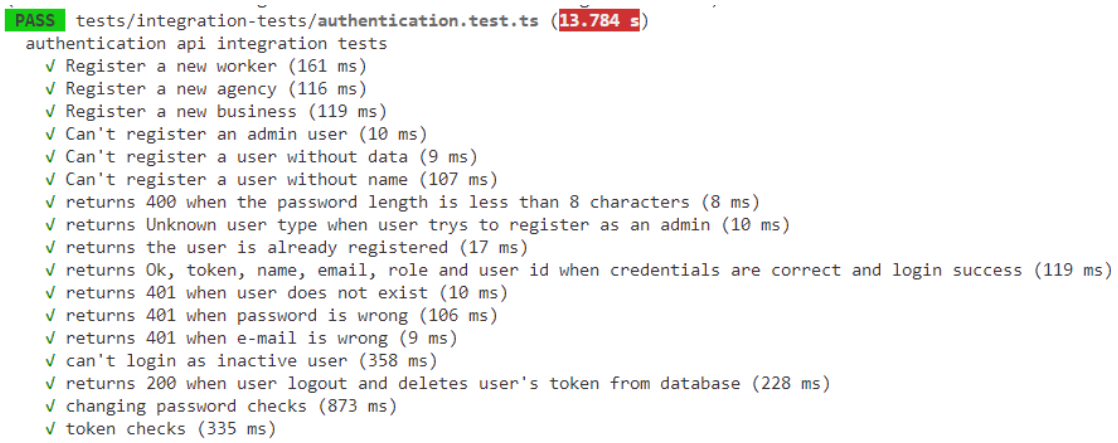

Figure 3. Successful integration tests were conducted on the authentication API, confirming the proper operation of each component within the authentication system.

By utilizing Jest's features, such as code coverage tracking and reporting, the study was able to identify areas of the code that required additional testing and improve the overall quality of the software. Furthermore, the Jest test report served as documentation of the testing process and provided evidence of the software's reliability and suitability for deployment.

As shown in Figure 4, the test report generated by Jest provided several benefits for the study. For instance, by using Jest, the tests were executed in a consistent and repeatable manner, ensuring that the test results were reliable and reproducible. Additionally, the report provided a clear overview of the test results, including the coverage metrics for function, branch, lines, and statement, indicating the percentage of code covered by the tests in each category.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	44.29	13.3	14.34	41.8	
innoBackend	92.64	55.55	80	93.84	
app.ts	92.64	55.55	80	93.84	39-40,93-94
innoBackend/controllers	88.04	65.38	76.92	88.21	
agreement.ts	100	100	100	100	
application.ts	100	100	100	100	
authentication.ts	94.44	89.47	100	94.11	108,111,123,310
feedBack.ts	100	100	100	100	
feeling.ts	100	100	100	100	
form.ts	100	100	100	100	
job.ts	100	100	100	100	
report.ts	100	100	100	100	
responsibility.ts	100	100	100	100	
topic.ts	100	100	100	100	

Figure 4. The report presented a comprehensive summary of the test outcomes, with coverage metrics for function, branch, lines, and statement, which accurately depicted the percentage of code covered by the tests in each category.

Throughout the testing process, the pursuit of achieving complete code coverage faced several challenges and limitations. The process of testing specific aspects of the code, including error handling and exception cases, presented difficulties. For instance, when testing the login process, difficulties arose in testing the case where a user's account had been blocked for security reasons, as shown in Figure 5. The code responsible for handling this scenario was difficult to test as it relied on specific conditions that were not easily replicable in a testing environment.

```
200     } else {
201       return res.status(403).json({
202         message: "This account has been blocked for security reasons",
203       });
204     }
```

Figure 5. The code that manages the testing of the scenario where a user's account is blocked due to security concerns was flagged as untested.

Similarly, during the registration process, the code handling errors and exceptions was challenging to test, as illustrated in Figure 6. The code that dealt with cases where a user document cannot be saved due to a database save

operation error or a validation error when creating a new user document was difficult to test.

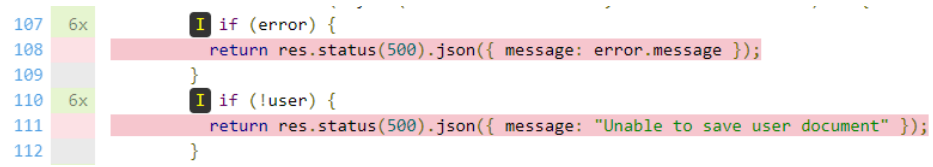

Figure 6. The testing of the code that manages errors and exceptions presented challenges, resulting in it being flagged as untested in the evaluation.

Furthermore, achieving complete code coverage was found to be time-consuming and resource-intensive, especially for larger codebases. Consequently, testing efforts had to be prioritized based on the most critical parts of the code.

Despite the encountered challenges, every effort was made to ensure that as much of the code as possible was covered by tests, ensuring the software's reliability and freedom from defects. The most critical parts of the code were tested through creating test cases, and various testing techniques were employed to increase code coverage. However, the constraints and limitations of the testing process made it infeasible to achieve complete code coverage.

4.4 Discussion

One of the key challenges in testing is setting up and maintaining test environments, so the study used an in-memory database for testing because it offers several advantages over traditional databases that can make the testing process more efficient and effective. First and foremost, testing with an in-memory database is generally faster than testing with a physical database. Since data is not being written or read to disk, tests can be executed more quickly. This is especially important for larger test suites, where the time to run tests can add up quickly. Another benefit of testing with an in-memory database is isolation. A fresh database can be started for each test, eliminating the risk of test data

interfering with other tests. This can help ensure that tests are more reliable and accurate, as they are only testing the specific functionality they were designed for. Additionally, consistency between tests can be ensured by using an in-memory database. With a physical database, data may be left over from previous tests or manual changes, which can make it difficult to ensure that each test is starting from the same baseline. Utilizing an in-memory database ensures that data is consistently in a reliable and stable state at the beginning of each test. In-memory databases have another advantage for testing, in that they are highly scalable. Unlike traditional databases, they are not limited by physical constraints and can manage large amounts of data and high levels of traffic without becoming overloaded or slowing down. This is particularly critical for testing applications that are designed to handle a massive volume of data or traffic. By using an in-memory database, developers can ensure that their applications are able to perform optimally under heavy load, which is essential for maintaining the stability and reliability of their software products. Finally, in-memory databases offer a remarkable degree of flexibility and customization. As they are specifically designed for testing, they can be effortlessly configured to accommodate the demands of diverse testing scenarios. For instance, they can be set up to mimic different forms of data, traffic, or user behaviour, thereby guaranteeing that tests are comprehensive and precise. This ability to adapt to different testing requirements makes in-memory databases a highly versatile tool in the software development process.

The results of this study highlight the critical role that testing plays in the software development life cycle. The successful implementation proved to be effective in establishing a suitable global environment for conducting tests, and the outcomes of the study were generally favourable, almost meeting the anticipated expectations. However, it is important to note that the implementation covered only two APIs due to the presence of a vast amount of untested code in the system version control, some of which was deemed redundant or irrelevant and some was non-functional or unreachable. This made it challenging to resolve issues and difficult to apply the testing process in those areas. In some instances,

this problem was overcome by removing part of the code that was either repetitive or insignificant.

The presence of long and convoluted code presented difficulties in both testing and rectifying issues, which required an extensive amount of time to address. The complexity of the code made it difficult to identify the root cause of the problems, slowing down the resolution process. Moreover, testing the code became more demanding, as errors or bugs may not have been immediately apparent due to the code's length and complexity. Therefore, it was crucial to write clear and concise code that was easy to test and maintain.

During the testing process, some challenges and limitations were encountered in attempting to achieve complete code coverage, that in some cases, it was difficult to test specific parts of the code, such as error handling or exception cases. For instance, the line "return res.status(500).json({message: error.message})" in a codebase was responsible for handling a 500 Internal Server Error response and sending a JSON payload back to the client. However, simulating this error condition during testing could be challenging, making it difficult to test this particular line of code and achieve complete code coverage. As a result, alternative testing strategies had to be devised, such as manual testing or code reviews, to ensure that these parts of the code were functioning correctly. Furthermore, a focus had to be placed on ensuring that the code performed correctly under a wide range of scenarios, including edge cases and unexpected situations, to improve the testing process's effectiveness.

The study emphasized that testing plays a crucial role in managing information security risks, that during the testing process, potential risks were identified, analysed, and addressed to avoid or eliminate them. The testing process involved identifying vulnerabilities and weaknesses in the system, such as software vulnerabilities, system weaknesses, or potential attack vectors that could be exploited by malicious actors. Risk analysis was performed to determine the potential impact of these risks on the system and the urgency required to address them. Once identified and analysed, appropriate measures were taken to mitigate

or eliminate the identified risks. These measures could include implementing security patches or updates, improving access control measures, or implementing additional security controls to address the identified risks.

4.5 Conclusion

In today's fast-paced software development environment, quality has become a vital element in the software development process. End-users have come to expect software that is reliable, user-friendly, and bug-free. Failing to meet these expectations can result in lost sales, reputation damage, and even legal action. Therefore, it is crucial to focus on testing early in the software design process.

When testing is integrated into the design process, it ensures that coding errors and bugs are identified early on and addressed before they grow exponentially, which makes them much harder to debug later in the development cycle. This approach can also reduce the time and cost of software development by detecting issues at the earliest possible stage, minimizing the need for rework or significant changes to the codebase.

A structured testing process that is followed from the beginning of the development cycle ensures that all aspects of the software are adequately tested, and any defects that are detected can be quickly and efficiently fixed. This results in higher-quality software that is more reliable and meets the needs of users.

Moreover, prioritizing testing in the software design process not only ensures quality but also helps to establish a culture of quality within development teams. By prioritizing quality, developers become more aware of the importance of testing, and it becomes an integral part of the development process. This culture of quality leads to a mindset where developers are continually looking for ways to improve the quality of the software they produce.

The incorporation of testing into the software development life cycle empowers developers to improve consistency and performance and create robust software. So, to ensure the ongoing success of this project, it is highly recommended to

prioritize and maintain a focus on testing in future development efforts. By dedicating time and resources to writing tests and continuously testing new features and resolved issues, developers can reduce the risk of delays and cost overruns and ensure the project progresses smoothly and the project can remain reliable and user-friendly, ultimately resulting in higher quality software.

5 Summary

Quality assurance plays a pivotal role in software development, as it ensures that the software adheres to the required specifications and delivers the intended functionality. Nonetheless, quality assurance involves extensive testing and analysis, which can be a complex and time-consuming process. To overcome these challenges, this study focused on enhancing the quality assurance process by leveraging powerful tools such as static analysis, TypeScript, and unit testing.

The study employed static analysis tools to detect various types of defects and issues, with the majority effectively addressed. However, limitations and difficulties stemming from design and methodology impacted the programmatic, quantitative, and temporal aspects of the remediation process. Static analysis tools are effective in detecting various types of defects and issues, but they have limitations when it comes to verifying whether a function fulfils its intended purpose. In addition, certain security vulnerabilities may not be easily detectable through automated means. Furthermore, unused files and unreachable data can slow down the process and potentially affect the accuracy of the results. Therefore, it is important to use static analysis tools alongside other methods to ensure thorough testing and analysis of the software.

The study also utilized TypeScript, which is known for its static typing capabilities that can help catch type errors early in the development process. However, the code was written without the proper use of types, with the type "any" being used extensively throughout the codebase. The coding practices used in the project lacked appropriate usage of types, with the "any" type being excessively utilized throughout the codebase. This lack of proper type annotations can result in a greater number of bugs and errors during program execution. Additionally, the TypeScript compiler is compelled to disable type checking for variables and returned values of call-backs and disregard any comments around them. Thus, to address such issues, it is crucial to leverage TypeScript's robust typing capabilities efficiently and consistently throughout the software development process.

Alongside static analysis and TypeScript, the study incorporated unit testing to guarantee the accuracy of each code component while operating in isolation. The testing process presented various challenges in attempting to achieve comprehensive code coverage, with certain code segments such as error handling and exception cases posing difficulty in testing. In response, alternative testing strategies were devised to ensure the correct functioning of these parts of the code. Additionally, the study focused on examining the software's behaviour under a variety of scenarios, including edge cases and unexpected situations, to enhance the effectiveness of the testing process.

Integration testing was also incorporated into the study to evaluate the software's behaviour in the context of its environment, examining the interactions between its individual components. This type of testing uncovered various defects and issues, which were addressed through appropriate remediation efforts. Through the combined use of unit and integration testing, the study was able to provide comprehensive coverage of the software's functionalities, improving the quality assurance process overall.

The study yielded promising results, indicating that the employed tools successfully improved the quality assurance process. However, certain difficulties and limitations, including those related to the design, methodology, and code complexity, impacted the overall effectiveness of the process. Therefore, it is crucial to continually enhance and refine the quality assurance process through the implementation of best practices and reliable tools to ensure the long-term maintainability and stability of the codebase.

References

[1] **R. Billairs, "What is static code analysis? Static analysis overview," Perforce, 10 February 2020. [Online]. Available: https://www.perforce.com/blog/sca/what-static-analysis#static. [Accessed 3 August 2022].**

[2] Encora, "Static Code Analysis: Types and How it Works," Encora, 18 July 2022. [Online]. Available: https://www.encora.com/insights/how-static-code-analysis-works. [Accessed 8 August 2022].

[3] Checkpoint, "What is Dynamic Code Analysis?," Check Point, 1994. [Online]. Available: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-dynamic-code-analysis/. [Accessed 4 June 2022].

[4] A. S. Gillis, "static analysis (static code analysis)," IBM, 1999. [Online]. Available: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis. [Accessed 29 June 2022].

[5] M. Ekaterina and K. Sergey, "How to choose a static analysis tool," 21 November 2021. [Online]. Available: https://pvs-studio.com/en/blog/posts/0884/. [Accessed 4 July 2022].

[6] N. C. Zakas, "ESlint Documentation," ESlint, 30 June 2013. [Online]. Available: https://eslint.org/docs/latest/. [Accessed 31 May 2022].

[7] J. Goldberg, "ASTs and typescript-eslint," typescript-eslint Doc, [Online]. Available: https://typescript-eslint.io/docs/development/architecture/asts/. [Accessed 17 December 2022].

[8] ESLint, "Configure Rules," ESLint, [Online]. Available: https://eslint.org/docs/latest/user-guide/configuring/rules. [Accessed 8 June 2022].

[9] Prettier, "What is Prettier?," Prittier, [Online]. Available: https://prettier.io/docs/en/index.html. [Accessed 9 June 2022].

[10] Prittier, "What is Prettier?," Prittier, [Online]. Available: https://prettier.io/docs/en/index.html. [Accessed 11 June 2022].

[11] ESLint, “Configuring ESLint,” ESLint, [Online]. Available: https://eslint.org/docs/latest/user-guide/configuring/. [Accessed 5 June 2022].

[12] Prettier, “Configuration File,” prettier Docs, [Online]. Available: https://prettier.io/docs/en/configuration.html. [Accessed 7 June 2022].

[13] Microsoft, “TypeScript for JavaScript Programmers,” Microsoft, 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. [Accessed 11 February 2023].

[14] Microsodt, “What is TypeScript,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/. [Accessed 3 March 2023].

[15] Microsoft, “TypeScript for the New Programmer,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 7 March 2023].

[16] Microsoft, “TypeScript for the New Programmer,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 7 March 2023].

[17] Microsoft, “TypeScript for the New Programmer,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 9 March 2023].

[18] Microsoft, “What is a tsconfig.json,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. [Accessed 15 March 2023].

[19] Microsoft, “Intero to the TSConfig Reference,” TypeScript, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/tsconfig. [Accessed 15 April 2023].

[20] Microsoft, “Enums,” Microsoft, 1st October 2012. [Online]. Available: https://www.typescriptlang.org/docs/handbook/enums.html. [Accessed 3 March 2023].

[21] JavaTpoint, “Unit Testing,” JavaTpoint, 2011. [Online]. Available: https://www.javatpoint.com/unit-testing. [Accessed 8 March 2023].

[22] Tutorialspoint, “Unit Testing,” Tutorialspoint, 2006. [Online]. Available: https://www.tutorialspoint.com/software_testing_dictionary/unit_testing.htm. [Accessed 7 October 2022].

[23] Javatpoint, “Integration testing,” Javatpoint, 2011. [Online]. Available: https://www.javatpoint.com/integration-testing. [Accessed 17 October 2022].

[24] R. Kuldeep, “Integration Testing,” ArtOfTesting, 29 May 2021. [Online]. Available: https://artoftesting.com/integration-testing. [Accessed 5 October 2022].

[25] j. T. point, “System Testing,” java T point, 2011. [Online]. Available: https://www.javatpoint.com/system-testing. [Accessed 13 December 2022].

[26] A. S. Gillis, “user acceptance testing (UAT),” TechTarget, 1999. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/user-acceptance-testing-UAT. [Accessed 10 April 2023].

[27] Lambdatest, “Jest Tutorial: Complete Guide to Jest Testing,” Lambdatest, 2017. [Online]. Available: https://www.lambdatest.com/jest. [Accessed 18 September 2022].

[28] K. Jeremy, “Comparing the best Node.js unit testing frameworks,” Logrocket, 13 May 2022. [Online]. Available: https://blog.logrocket.com/comparing-best-node-js-unit-testing-frameworks/. [Accessed 28 September 2022].

[29] M. Waweru, “Beyond API testing with Jest,” Circleci Blog, 30 September 2022. [Online]. Available: https://circleci.com/blog/api-testing-with-jest/#c-consent-modal. [Accessed 29 November 2022].

[30] K. Jeremy, “Comparing the best Node.js unit testing frameworks,” Logrocket, 13 May 2022. [Online]. Available: https://blog.logrocket.com/comparing-best-node-js-unit-testing-frameworks/. [Accessed 29 Spetember 2022].

[31] Facebook, “JEST,” Facebook, [Online]. Available: https://jestjs.io/. [Accessed 5 October 2022].

[32] npm, “ts-jest,” npm, 2014. [Online]. Available: https://www.npmjs.com/package/ts-jest. [Accessed 6 October 2022].