

Prathibha Bollu

# ENSURING MAINTAINABILITY IN REACT WEB APPLICATIONS

Master of Science  
Faculty of Information Technology and Communication Sciences  
Supervisors: Prof. Kari Systä, Prof. Terhi Kilamo  
November 2024

## ABSTRACT

Prathibha Bollu: ENSURING MAINTAINABILITY IN REACT WEB APPLICATIONS

Master of Science

Tampere University

Master's Degree Program in Software, Web and Cloud

November 2024

---

The goal of this research is to analyse maintainability in the context of React web apps and propose methods to ensure it. As the complexity of these systems increases, maintainability becomes increasingly essential, often becoming more costly than the development phase. However, it is frequently overlooked at the outset of a project.

Many factors contribute to the maintainability of software, but this thesis focuses on how specific features of React components—like their size, complexity, and nesting patterns impact it. By studying these aspects and their connection to bug density, this research offers a clear understanding of which structural choices help keep React applications stable and easier to maintain over time.

This study provides an empirical analysis, examining component metrics and how maintenance actions are related, focusing on component size, complexity, and nesting depth within open-source React applications. Using bug density as an outcome-based metric for maintainability, the research explores how specific structural attributes of components impact long-term stability. Through correlation and regression analysis, this approach aims to identify structural factors most associated with defect rates and proposes concrete recommendations for designing maintainable React components.

The findings of this study are expected to reveal the impact of component size, complexity, and nesting depth on defect rates, offering actionable insights for developers. By adopting specific design practices, developers may be able to mitigate technical debt, enhance debugging ease, and support scalable, sustainable application development in React over time.

Keywords: Maintainability, react web applications, complexity, component size, nesting patterns, empirical analysis, open-source React applications.

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

## Use of AI in Thesis

I have utilized AI tools in my thesis: Yes

**The AI tools utilized in my thesis and their purposes are described below:**

### AI Tools Used:

1. SciSpace
2. QuillBot
3. Grammarly
4. ChatGPT

### Purpose of Using AI Tools:

1. **Language Enhancement:** Tools such as QuillBot and Grammarly were essential in improving the language of my thesis. QuillBot assisted in rephrasing and rewording sentences to enhance fluency and clarity, while Grammarly ensured the text was grammatically sound and free from punctuation mistakes. Together, these tools improved the overall readability and professionalism of the thesis.
2. **Research Support and Idea Development:** SciSpace and ChatGPT played vital roles in helping me explore and integrate research. SciSpace helped to quickly comprehend complex academic papers by offering summaries and context, while ChatGPT assisted with brainstorming, generating ideas, and explaining difficult concepts. It also helped in structuring and organizing the content, particularly during the initial writing phases.
3. **Proofreading and Final Edits:** In the final stages, Grammarly was instrumental in ensuring that the thesis was grammatically correct and polished stylistically. It helped refine the final draft by identifying minor errors that might have been overlooked.

### Sections Where AI Tools Were Utilized:

1. SciSpace was employed to streamline the Literature Review process. It helped me explore and understand academic papers by providing summaries and contextual explanations.
2. QuillBot and Grammarly were used throughout the document to enhance the content, improve sentence structure, and guarantee grammatical accuracy and proper punctuation.
3. ChatGPT was particularly valuable in refining the Introduction and Discussion sections, improving clarity, flow, and the articulation of complex ideas.

## PREFACE

The topic of maintainability in software was initially recommended by my professor, Professor Kari Systä, who recognized its significance and potential for further exploration. As I immersed myself in the research and reviewed existing literature, my interest grew, especially in understanding maintainability challenges within the React framework, given its prominence in web development. This led to a focused investigation on practices that can enhance the maintainability of React applications.

The formal work on this thesis began in my fourth semester, aligning with my academic goal to contribute valuable insights to the field and my personal goal of deepening my expertise in React. I would like to express my sincere gratitude to Professor Kari Systä for his unwavering support and valuable guidance throughout the research and writing process. I also extend my thanks to Professor Terhi Kilamo, who served as the examiner for this thesis.

Tampere, 21st November 2024

Prathibha Bollu

## CONTENTS

1.	Introduction . . . . .	1
2.	Related work and Literature Review . . . . .	3
2.1	General Overview of Maintainability . . . . .	3
2.2	Distinguishing Maintainability from Maintenance . . . . .	4
2.3	Maintainability in React Applications . . . . .	4
2.4	Factors Impacting Maintainability in React Applications . . . . .	5
2.5	Recent Improvements in React Maintenance Practices . . . . .	6
3.	The Evolution of Maintainability in Software Development. . . . .	8
3.1	Introduction to the Evolution of Maintainability . . . . .	8
3.2	Changing Maintainability Needs Throughout the Software Project . . . . .	8
3.3	Advanced Tools and Techniques for Maintaining Large-Scale React Applications . . . . .	9
4.	Research Questions and Methods . . . . .	11
4.1	Introduction . . . . .	11
4.2	Research Question . . . . .	11
4.3	Methodology for Research Question . . . . .	12
4.3.1	Selection of Open-Source React Projects . . . . .	12
4.3.2	Data Collection . . . . .	12
4.3.3	Statistical Analysis . . . . .	13
4.4	Expected Outcomes . . . . .	14
5.	Implementation . . . . .	15
5.1	Visualization and Metrics. . . . .	15
5.1.1	Scatter Plots . . . . .	15
5.1.2	Correlation Heatmaps . . . . .	18
5.1.3	Cross-Project Insights . . . . .	20
5.2	Regression Analysis . . . . .	22
5.2.1	Explanation of Key Regression Terms . . . . .	22
5.2.2	Project-Specific Results . . . . .	23
5.2.3	Cross-Project Insights from Regression . . . . .	24
5.3	Summary of Findings . . . . .	25
5.4	Developer Recommendations Based on Findings . . . . .	26
6.	Conclusion . . . . .	28

## LIST OF FIGURES

5.1	Scatter plot illustrating the relationship between Lines of Code (LOC) and Bug Density in the all three projects. . . . .	16
5.2	Scatter plot illustrating the relationship between Max Complexity and Bug Density in the all three projects. . . . .	16
5.3	Scatter plot illustrating the relationship between Nesting Depth and Bug Density in the all three projects. . . . .	17
5.4	Correlation heatmap illustrating the relationships between component metrics and bug density in the Ant Design project. . . . .	18
5.5	Correlation heatmap illustrating the relationships between component metrics and bug density in the React Icons project. . . . .	19
5.6	Correlation heatmap illustrating the relationships between component metrics and bug density in the Semantic-UI project. . . . .	19

## LIST OF TABLES

5.1	Regression results for Ant Project . . . . .	23
5.2	Regression results for React Icons Project . . . . .	24
5.3	Regression results for Semantic Project . . . . .	24

# 1. INTRODUCTION

In the ever-changing world of web development, the creation of scalable and maintainable React applications is crucial to meet the evolving demands of users and technological advancements. React, the JavaScript Library developed by Facebook, has revolutionized front-end development with its component-based architecture and efficient virtual DOM. Despite its strengths, as React applications scale, challenges in maintainability often emerge, impacting both code quality and team efficiency.

Overly complex components, tightly coupled hierarchies, and overuse of component state management can lead to technical debt, architectural weaknesses, and performance bottlenecks. These challenges become more pronounced in collaborative environments where differences in coding styles and knowledge levels exacerbate the difficulty of managing the codebase over time.

To address these hurdles, developers must prioritize React-centric practices such as refactoring for improved code modularity, readability, and adherence to best practices. Additionally, tools like ESLint play a crucial role in enforcing consistent coding conventions, promoting maintainability, and reducing errors. Fostering a culture of knowledge sharing, testing, and continuous integration ensures that React applications can evolve sustainably.

This research delves into specific ways in which component design, coding conventions impact the maintainability of React applications. Through statistical analysis, industry best practices, this study aims to provide actionable recommendations to help individuals and teams build React applications that are robust and adaptable.

## **Motivation**

As a front-end developer, I am inherently driven to delve into the underlying mechanics of React development. Understanding how component design decisions, adherence to coding conventions, and other factors affect the long-term maintainability and scalability of applications fuels my interest in this research. Through firsthand experience, I have witnessed the direct impact of these factors on project health. Thus, this research aims to bridge the gap between theoretical best practices and real-world application, providing valuable insights for developers navigating the complexities of React development.

## **Goals**

In this research, our main goal is to provide practical insights that developers can use to improve their React projects. We're diving deep into things like how component size and coding rules affect how easy it is to maintain a React application. Our aim is to give developers straightforward suggestions based on real data so they can make the React applications better. Additionally, I plan to implement these findings first and then proceed with writing the thesis. This approach will provide me with greater confidence in presenting the results effectively and comprehensively in the final paper.

## 2. RELATED WORK AND LITERATURE REVIEW

### 2.1 General Overview of Maintainability

Maintainability, an essential component of software quality, is frequently overlooked throughout the early stages of the software development lifecycle. While quality is typically emphasized as a critical need for software products, maintainability receives less attention until the maintenance phase. However, it is during this later phase that the challenges of maintaining and evolving software systems become the most apparent. According to research, the maintenance phase can account for a large amount of overall ownership costs, sometimes exceeding the initial development cost (Chen and Huang, 2009).

Addressing maintainability issues early in the software development process is essential for reducing the challenges and costs of maintenance (Behnamghader and Boehm, 2019). Developers can considerably reduce overall maintenance costs and complexity by understanding and proactively resolving common maintainability challenges during the design and implementation phases. This proactive approach involves introducing coding methods, design patterns, and architectural concepts that emphasize code clarity, modularity, and flexibility (Boehm and Papaccio, 1988).

Integrating maintainability into the development process promotes a culture of continual improvement and sustainability. It enables developers to predict and adjust to future changes, improvements, and updates more effectively and efficiently. By establishing a solid foundation of maintainable code, businesses may streamline the maintenance process, reduce technical debt accumulation, and ensure the long-term durability of their software products (Behnamghader and Boehm, 2019; Viljanen, 2015).

While maintainability is often not prioritized in the initial phases of many software development projects, its importance grows during the maintenance phase. However, depending on the type of software and the development approach, some teams do consider maintainability from the beginning. Organizations may optimize resources, improve software quality, and maximize the return on investment by addressing maintainability early in the development lifecycle.

## 2.2 Distinguishing Maintainability from Maintenance

In software engineering, "maintainability" and "maintenance" are closely related but distinct concepts. Maintainability refers to the ease with which a software system can be modified to fix defects, improve performance, or adapt to changes. It is an inherent attribute of the software, determined by its design, code structure, and documentation. High maintainability allows for efficient and low-risk modifications, facilitated by modularity, clear coding practices, and thorough documentation (Dinku, 2022).

Maintenance, on the other hand, is the process of performing those modifications after the software has been deployed. This includes fixing bugs (corrective maintenance), adapting the software to new environments (adaptive maintenance), improving performance (perfective maintenance), and making changes to prevent future issues (preventive maintenance) (Dinku, 2022).

While maintenance involves the ongoing work to keep the software functional, maintainability is what determines how easily and cost-effectively this work can be done. A highly maintainable system reduces the time and effort needed for maintenance activities, thus ensuring the long-term sustainability of the software.

## 2.3 Maintainability in React Applications

Maintaining a React application requires a variety of ways to keep the codebase understandable, adaptive, and scalable throughout its lifecycle. React, a popular JavaScript toolkit for creating user interfaces, presents unique challenges and considerations for maintainability.

Maintainability in React revolves around how easily developers can explore, alter, and expand the codebase. Using React's declarative and component-based architecture, developers may divide complicated user interfaces into smaller, reusable components (Facebook, 2023). This modular approach improves code structure and maintainability by simplifying functionality and minimizing code complexity (Pavić and Brkić, 2021).

Maintainability in React encompasses more than just code structure; it also includes tooling, documentation, and adherence to coding standards. Creating explicit code conventions, directory structures, and documentation processes encourages consistency and collaboration among development teams (Misra, 2005). Furthermore, applying code analysis and formatting tools such as ESLint guarantees coding standards are followed and helps detect maintainability problems early in the development cycle (ESLint, 2016).

Moreover, React apps' overall maintainability depends on their performance being optimized (Bytes, 2021). Performance bottlenecks can worsen user experience and require more maintenance work. Examples of this include inefficient and repeated rendering.

Developers can improve the efficiency and responsiveness of React apps, which will strengthen their maintainability, by applying performance optimization techniques and following industry best practices (Bytes, 2021; Pavić and Brkić, 2021).

To put it simply, maintaining a React application necessitates a thorough strategy that balances coding conventions and best practices with performance, modularity, and code clarity. By emphasizing maintainability throughout the development process and adopting proactive strategies for code organization, documentation.

## 2.4 Factors Impacting Maintainability in React Applications

In the realm of React development, several factors exert significant influence on the long-term maintainability of applications. To ensure that their React projects remain manageable and of high quality, developers must recognize these variables and take appropriate action. Now, let us examine the important factors that influence the overall maintainability of React apps:

- **Architecture and Organization of Components:** Maintainability of a React application is heavily dependent on how its components are organized and structured. Modularity, reusability, and readability are enhanced by a well-structured component hierarchy. In contrast, complex or closely connected component structures can make maintenance more difficult and complicated (Facebook, 2023).
- **Complexity of State Management:** React apps that have dynamic user interfaces and complex data flows will not be maintainable without effective state management. The choice of state management tools, whether external libraries like Redux or the built-in capabilities of React, has a significant impact on how maintainable the application is. By encouraging a clear separation of interests, a well-defined state management technique facilitates data flow, reduces side effects, and improves code maintainability (Redux, 2021).
- **Code Quality and Consistency:** React codebases' quality and readability depend heavily on its members' consistent adherence to coding conventions, standards, and best practices. When it comes to code linting and formatting, using tools such as ESLint guarantees code consistency, spots possible problems early on, and promotes maintainability by minimizing errors and inconsistencies.
- **Best Practices for Documentation:** Detailed documentation is an essential tool for comprehending the functions, uses, and purposes of the various modules and components that make up a React application. Code that is well documented makes it easier for developers to collaborate, share knowledge, and carry out future maintenance. Meaningful comments also help with code maintenance and modification by clarifying the reasoning behind particular code decisions.

- **Performance Optimization:** Maintaining the maintainability of React applications requires continuous performance optimization. Reducing needless re-renders and streamlining rendering operations are two ways to address performance bottlenecks and ensure the responsiveness and scalability of applications. Ignoring performance issues can make maintenance more difficult and jeopardize the application's long-term viability (Facebook, 2023).

## 2.5 Recent Improvements in React Maintenance Practices

React maintenance methods have significantly improved in the last few years due to technological developments, community-driven initiatives, and an increasing focus on developer experience and code quality. React apps are now more resilient, scalable, and maintainable.

Tooling and development workflows are one area where there has been a noticeable improvement. Strong tools and libraries to improve code quality and expedite development processes have emerged in the React ecosystem. For instance, tools like Next.js, Gatsby.js, and Create React App (CRA) give developers boilerplate configurations and scaffolding to enable quick application creation while following best practices for code organization and optimization (Facebook, 2023).

React application maintainability has also improved significantly as a result of developments in state management technologies. Libraries that reduce boilerplate code and complexity, including Recoil and Redux Toolkit, provide more user-friendly and effective means of managing application state. Codebases that are easier to maintain and scale are produced by these state management solutions, which give developers reliable rules for controlling data flow and synchronizing state across components (Pavić and Brkić, 2021).

React hooks have also completely changed how developers create and maintain stateful logic in functional components. Hooks that support a more declarative and composable method of controlling component state and side effects include `useState`, `useEffect`, and `useContext`. Developers can increase code modularity and reusability, which results in more understandable and maintainable code, by enclosing logic behind custom hooks.

Improved code quality and maintainability in React apps have also been facilitated by advancements in static typing with technologies such as TypeScript and Flow (Chen and Huang, 2009). Improved code description, better developer tooling support, and early mistake detection are all made possible by static typing. Developers may work together more productively in larger codebases, refactor code with confidence, and detect problems at compile time by utilizing static typing.

In addition, a culture of cooperation and information exchange has been promoted within

the React community by community-driven projects like the RFC (Request for Comments) procedure and the adoption of best practices. As a result of everyone's combined efforts, patterns, principles, and suggestions for building better maintainable React code have been discovered and shared.

In recent years, the React ecosystem has seen remarkable advancements that have transformed how developers maintain their applications. The introduction of hooks, like `useState` and `useEffect`, has streamlined the way state and side effects are managed, allowing for more intuitive and concise component code (Singh and Srinivasan, 2024). Additionally, integrating TypeScript into React projects has become a game-changer. By catching errors at compile time, TypeScript enhances code reliability and reduces the effort needed for bug fixes, which directly contributes to better maintainability (Bogner and Merkel, 2022).

State management has also evolved significantly with the adoption of tools like Redux Toolkit. These tools simplify data handling and minimize boilerplate code, making even complex applications easier to manage (Kuznetsov and Kuznetsova, 2022). Beyond that, enforcing consistent coding standards with tools like ESLint has proven essential for early detection of potential issues, helping maintain high code quality throughout the project lifecycle (Editorial, 2024). Together, these innovations have led to more efficient development workflows, cleaner codebases, and an overall improved experience for developers, making React applications more maintainable and scalable than ever before.

## **3. THE EVOLUTION OF MAINTAINABILITY IN SOFTWARE DEVELOPMENT**

### **3.1 Introduction to the Evolution of Maintainability**

Maintainability, as discussed earlier in Chapter 2, is a critical software quality attribute and affects how easily software can be modified to correct defects, improve performance, or take up the requested changes if needed. Even though the importance of maintainability is highlighted during the design stage, its increasing criticality becomes apparent as the software system starts growing or evolving in size and complexity over time. This chapter discusses maintainability needs in the software life cycle, especially highlighting advanced tools and methodologies that assist in maintainable software design and development in React applications.

### **3.2 Changing Maintainability Needs Throughout the Software Project**

In the early stages of a software project, the main focus is to establish a solid architectural foundation that supports long-term maintainability. As stated in Section 2.4, good choices in terms of modular design and extensive documentation are part of the major contributors to a maintainable code base from the start (Rosene and Connolly, 1981). With React applications, the built-in component-based architecture enforces reusability and separation of concerns by its nature (Phan, 2019).

As time progresses, complexity usually tends to increase with the code base of a software project, ushering in new challenges that involve preserving software. For instance, a large React application enables one to mishandle state management between components quite easily, leading to tightly coupled and less maintainable code (Dinku, 2022). The strength of tools like Redux and Context API calls for great care to avoid falling into these traps; it is important to preserve a clean and maintainable structure of an application (Tran, 2023).

In later phases of the software life cycle, particularly in mature projects, accumulated technical debt becomes more of a matter on its own. Continuous refactoring and per-

formance tweaking are critical. In a React application, tools like ESLint and TypeScript are invaluable for keeping code quality up to par and assuring the maintainability of the software as it gets more complex (Elmidaoui et al., 2019). The system does require regular code reviews, refactoring, and performance optimization in order to ensure that the system remains maintainable as it scales up (Khuat, 2018).

### 3.3 Advanced Tools and Techniques for Maintaining Large-Scale React Applications

To retain high levels of maintainability in large-scale React applications, developers must adopt tools and techniques that are advanced in nature and go beyond basic best practices. This section will first introduce some of the most effective tools and practices, specifically tailored to large and complex React projects.

- **Static Analysis Tools:** These include static code analysis tools such as SonarQube and Code Climate, among others that offer deep insights into code quality and maintainability. Such tools help identify potential issues early enough—things like complex functions or poorly structured code that are common in large React applications. Through early detection of these issues, developers can prevent such issues from cropping up in the future and hence stop major maintainability problems from erupting in the code base (Baggen et al., 2012).
- **Good Refactoring Practices:** Refactoring must be applied continuously so as to keep the technical debt under control and thus keep the source code clean and modular. In React applications, that would frequently mean simplifying state management, breaking large components apart into smaller ones which are easy to manage, and leveling down component hierarchies. These practices help keep the codebase maintainable as the application grows (Bogner et al., 2017).
- **Performance Optimization Techniques:** As already mentioned in Chapter 2, performance is one of the contributors to the maintainability of any program. In fact, tools such as React Profiler are used by developers to identify which parts have performance bottlenecks that are difficult to maintain. Thus, techniques around memoization, lazy loading, and code splitting become very important to keep the performance optimized and large React projects maintainable (Fedosejev, 2015; Poudel, 2023).
- **Automated Testing:** High test coverage is very important to keep the codebase stable and maintainable. Tools like Jest and Cypress are commonly used to ensure that changes in React projects do not introduce new bugs, which would become critical when the project scales. Automated testing contributes to maintainability because developers can execute refactoring work assured that the existing behaviour

is safeguarded by tests (Han, 1997).

- **Continuous Integration/Continuous Deployment (CI/CD):** Activating CI/CD pipelines ensures that changes made in the code will be automatically passed through the testing and deployment process, hence reducing the risk of maintainability issues. It is particularly useful in cases where a large team is handling complex React applications to maintain consistency and identify problems upfront.

## 4. RESEARCH QUESTIONS AND METHODS

### 4.1 Introduction

This chapter describes the research questions that form the foundation of this study and details the methodologies used to address them. The main objective of this research is to gather empirical evidence on the factors that influence maintainability in React applications, specifically focusing on component size, complexity, and nesting patterns. This analysis aims to provide insights and recommendations to assist developers in improving maintainability through better design practices in component-based architectures.

### 4.2 Research Question

**Research Question:** How do component size, complexity, and nesting practices impact the maintainability of React projects?

React applications are typically component-based, and the structural design of these components is critical for long-term maintainability. In this study, maintainability is assessed using bug density—a practical metric indicating the number of bugs relative to the component's size. This research investigates three main factors that could directly or indirectly affect ease of modification and defect frequency within components: component size, complexity, and nesting depth.

#### Objective

- To evaluate the relationship between component characteristics (size, complexity, nesting depth) and bug density.
- To develop recommendations on component design practices that promote maintainability.

#### Significance for Developers

Understanding how component structure affects maintainability can guide React developers toward best practices that reduce technical debt and improve code stability and adaptability.

## 4.3 Methodology for Research Question

### 4.3.1 Selection of Open-Source React Projects

To ensure robust and applicable findings, this analysis includes React projects from well-maintained, open-source repositories chosen based on:

- **Size and Structure:** Projects of varying sizes were selected to represent a spectrum of maintainability challenges across small, medium, and large codebases.
- **Consistent Maintenance:** Only projects with a history of regular maintenance and a significant number of contributors were selected, ensuring the findings reflect well-established development practices.

#### Chosen Projects

1. **Ant Design (Large project):** Known for its enterprise-level UI components, this project has a complex, modular codebase suitable for studying maintainability at scale.
2. **Semantic UI React (Medium project):** Offers balanced complexity, representing mid-sized applications with moderate structural complexity.
3. **React Icons (Small project):** A minimalistic library focused on SVG icons, providing insight into maintainability in smaller codebases.

### 4.3.2 Data Collection

Data collection involved obtaining detailed structural and maintenance metrics from each project, focusing on elements that directly relate to component maintainability.

#### Component Metrics Collection

- **Lines of Code (LOC):** This simply counts the actual lines of code in a component, ignoring empty lines and comments. It gives us an idea of the component's size, where larger components may be harder to maintain.
- **Cyclomatic Complexity:** This metric evaluates how complicated the code is. It looks at how many different paths the code can take based on decisions (like if statements and loops). The more paths there are, the harder it is to understand and maintain the code. We used a tool called ESLint to automatically check this complexity.

$$\text{Cyclomatic Complexity} = E - N + 2P$$

where:

- $E$  represents the number of edges in the control flow graph,
  - $N$  denotes the number of nodes in the control flow graph, and
  - $P$  is the number of connected components (usually 1 for a single function).
- **Maximum Nesting Depth:** This metric checks how deeply nested the code is (like layers within layers). For example, if you have several nested if statements or loops, it can become hard to read and debug. We used ESLint to find the deepest level of nesting in each component.

The data collection process used an automated script to scan each project directory, apply ESLint rules, and record LOC, complexity, and nesting depth for each component. Non-component files (e.g., utility files, configuration files) were excluded from the analysis to maintain a focus on React components alone.

### Maintenance Metrics - Bug Density Measurement

- **Why Bug Density?** Our research focuses on things like the size, complexity, and nesting of React components to see how they affect maintainability. But just looking at these design aspects alone doesn't give us the full picture. For example, a large or complex component might seem like a problem, but it could still be easy to manage if well-tested. On the other hand, a simple component can become a headache if it's frequently changed or updated without coordination. That's where Bug Density comes in—it tells us how often bugs actually appear, giving us a real sense of what's causing issues in the code. It helps us understand the practical impact of our design choices and shows us which parts of the code need extra attention.
- **Filtering Commit Data:** We collected commit messages from the project's version control system and filtered them using keywords like "fix," "bug," and "defect." This helped us identify only the defect-related changes.
- **Calculation:** Bug Density for each component was calculated using :

$$\text{Bug Density} = \frac{\text{Number of Bug-Related Commits}}{\text{Lines of Code (LOC)}}$$

### 4.3.3 Statistical Analysis

To interpret how each structural metric relates to bug density, we used multiple statistical methods:

- **Scatter Plots:** These provide a visual representation of how each structural metric (LOC, complexity, and nesting depth) correlates with bug density, allowing for an intuitive understanding of any observable trends.

- **Correlation Analysis (Heatmaps):** Correlation heatmaps were created to assess associations between component metrics and maintenance activities. These heatmaps illustrate the strength of the relationships, highlighting factors that are likely to affect defect rates.
- **Regression Analysis:** A multiple linear regression analysis was conducted to measure how much each metric (like component size, complexity, and nesting depth) contributes to bug density. Unlike simple correlation, which shows only general relationships, regression allows us to see the specific impact of each metric while keeping the others constant. This helps us understand more clearly how each component characteristic affects maintainability on its own.

#### 4.4 Expected Outcomes

The analysis aims to:

- **Identify Key Structural Drivers:** Determine which component metrics (size, complexity, nesting depth) most significantly impact bug density.
- **Develop Guidelines for Component Design:** Offer recommendations on ideal component sizes, acceptable complexity thresholds, and best practices for nesting depth that can enhance maintainability in React applications.

These findings aim to guide developers in adopting structural practices that reduce bug density, fostering maintainable, stable, and scalable React applications.

## 5. IMPLEMENTATION

### Introduction

RQ1: How do component size, complexity, and nesting practices impact the maintainability of React projects?

Maintaining software that can easily evolve is essential in today's software development landscape, where stability and adaptability are constant concerns. In this study, bug density—an outcome-based metric—is used to evaluate maintainability. Bug density represents the number of defects per code unit, offering a clear picture of stability. A higher density indicates challenges in maintaining and expanding the software, often resulting in increased upkeep costs.

### 5.1 Visualization and Metrics

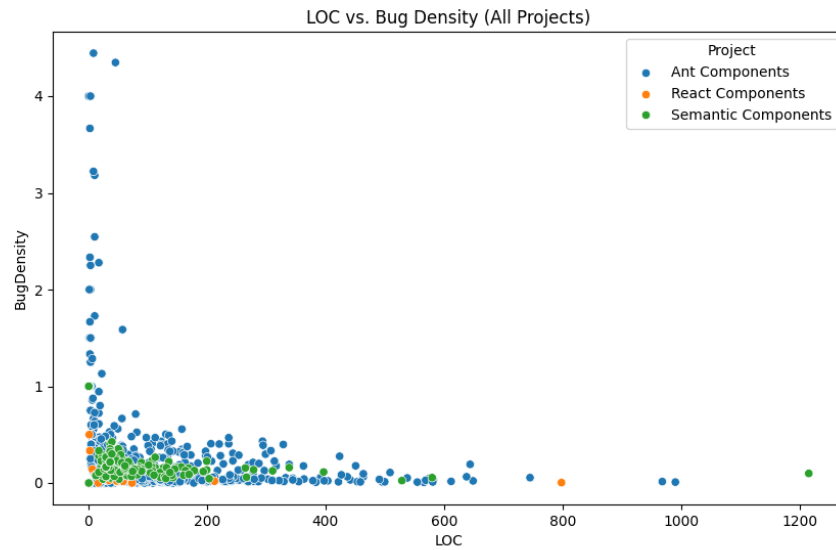
To analyze the relationship between component structure and maintainability, we visualized key metrics: component size, complexity, and nesting depth alongside bug density. This approach aimed to uncover patterns showing whether larger, more complex, or deeply nested components exhibit higher bug densities. Such insights could inform best practices for designing maintainable code.

#### 5.1.1 Scatter Plots

Scatter plots were chosen to visually assess how Lines of Code (LOC), Maximum Complexity, and Maximum Nesting Depth correlate with Bug Density. By doing so, we gain a clear picture of how component size and complexity may impact the codebase stability in three distinct projects: Ant Design, React Icons, and Semantic UI.

#### Relationship Between Lines of Code and Bug Density

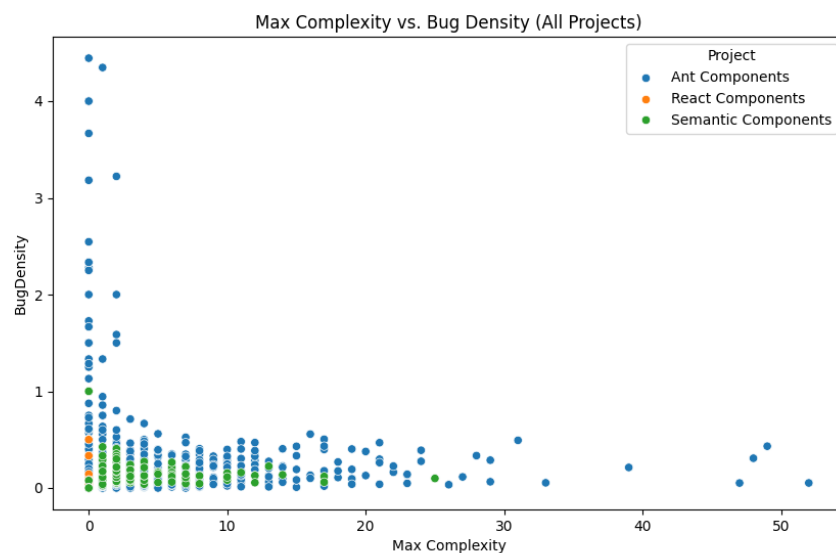
This plot explores the relationship between component size (measured in LOC) and bug density. Across all three projects, smaller components tend to have high bug densities. In particular, in Ant Design, there are noticeable outliers where small components exhibit extremely high bug density, indicating that frequent updates without sufficient testing may



**Figure 5.1.** Scatter plot illustrating the relationship between Lines of Code (LOC) and Bug Density in the all three projects.

contribute to defects, regardless of component size. This suggests that structural modularity, paired with regular testing, could help manage bug rates effectively, even for larger components.

### Relationship Between Max Complexity and Bug Density

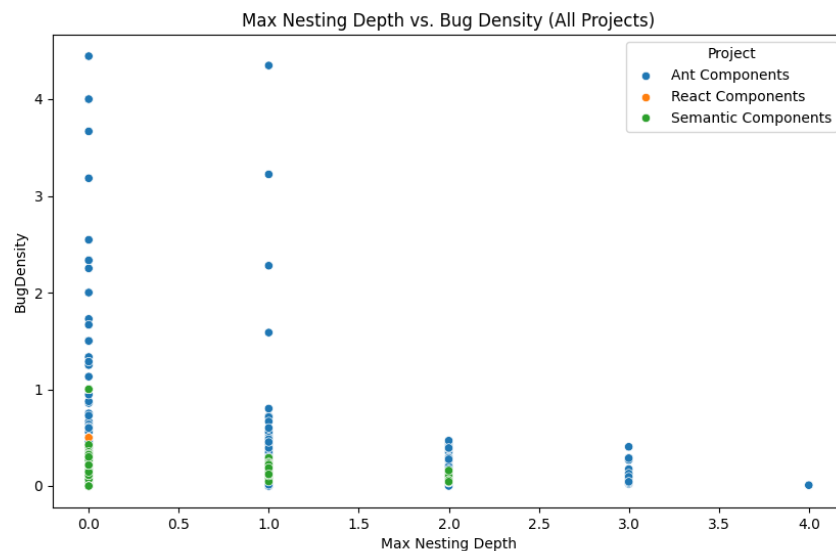


**Figure 5.2.** Scatter plot illustrating the relationship between Max Complexity and Bug Density in the all three projects.

This plot investigates whether components with higher cyclomatic complexity—an indicator of logical complexity—also experience higher bug densities. Within Ant Design,

several lower to moderate level complexity components indeed display elevated bug density, underscoring the need for targeted testing and refactoring to manage complexity and uphold maintainability. In contrast, React Icons shows components with moderate complexity levels and relatively low Bug Density, suggesting effective management of logical complexity. The scatter plot clearly shows that React Icons contains fewer high-complexity outliers than Ant Design, with the majority of components concentrated at moderate complexity and low defect levels. Semantic UI, on the other hand, uses only low-complexity components (0 complexity level), indicating a very simple component design with little logical complexity. This visual evidence, together with the reduced average Bug Density for React Icons and Semantic UI, suggests that simplified component structures and consistent testing contribute to effective complexity management."

### Relationship Between Max Nesting Depth vs. Bug Density



**Figure 5.3.** Scatter plot illustrating the relationship between Nesting Depth and Bug Density in the all three projects.

This plot examines the relationship between the nesting depth of components and bug density to see if deeply nested structures contribute to higher defect rates. For Ant Design, there's a clear pattern of increased bug density in components with deeper nesting levels, hinting that structural depth can introduce additional maintenance challenges. On the other hand, React Icons and Semantic UI demonstrate minimal correlation between nesting depth and bug density, likely due to simpler component hierarchies. These findings suggest that minimizing deep nesting may improve maintainability, especially in complex projects.

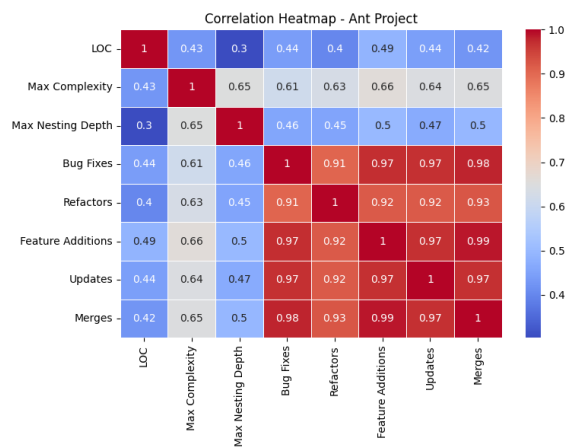
Overall, these scatter plots indicate that component structure impacts maintainability differently across projects. Design practices, modularity, and testing protocols play substan-

tial roles in mediating these effects, highlighting that an adaptable approach to component structure is essential for effective project maintenance.

## 5.1.2 Correlation Heatmaps

To deepen the understanding of relationships between component structure and maintainability, we generated correlation heatmaps for each project. These heatmaps illustrate the strength of association between various component metrics, such as Lines of Code (LOC), Maximum Complexity, and maintenance activities (e.g., Bug Fixes). Through these visualizations, we identified factors most likely to contribute to defect rates.

### Ant Design Correlation Heatmap



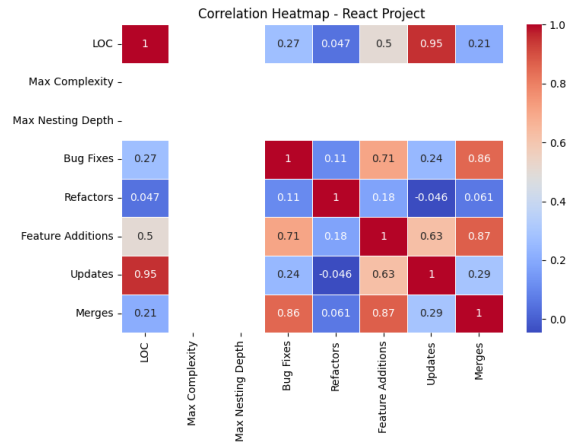
**Figure 5.4.** Correlation heatmap illustrating the relationships between component metrics and bug density in the Ant Design project.

For Ant Design, there's a notable positive correlation between Bug Fixes, Refactoring, and Feature Additions, implying that specific components consistently demand more maintenance effort. Additionally, there's a moderate correlation between LOC, Max Complexity, and Bug Fixes, suggesting that larger, more complex components tend to be more defect-prone. This trend highlights the importance of maintaining control over component size and complexity, especially within large, enterprise-grade applications where extensive functionality can heighten the risk of defects.

### React Icons Correlation Heatmap

In the React Icons project, LOC strongly correlates with Bug Fixes and Updates, indicating that larger components often require more maintenance. Interestingly, the weaker correlation with Max Complexity suggests that React Icons' modular architecture aids in managing complexity. This insight implies that prioritizing manageable component sizes within a modular structure can reduce the influence of complexity on bug density, sup-

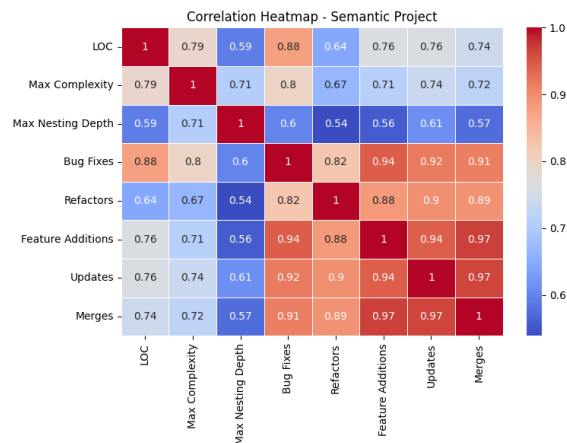
porting the maintainability of the project over time.



**Figure 5.5.** Correlation heatmap illustrating the relationships between component metrics and bug density in the React Icons project.

### Semantic UI Correlation Heatmap

For Semantic UI, the heatmap shows strong positive correlations between LOC, Bug Fixes, Refactors, and Updates, echoing patterns observed in the other projects. The moderate correlation with Max Complexity suggests that, while modularity supports maintenance, actively managing complexity is also essential. This finding underscores that complexity control is crucial for sustaining maintainability, as modularization alone may not sufficiently mitigate all maintenance challenges in projects with intricate structures like Semantic UI.



**Figure 5.6.** Correlation heatmap illustrating the relationships between component metrics and bug density in the Semantic-UI project.

### 5.1.3 Cross-Project Insights

Now that we have the correlation heatmaps for the three projects, let's break down the cause-and-effect relationships, mutual influences, and parallel factors based on the heatmap data.

#### 1. Cause-Effect Relationships Based on Correlation Data

##### **Max Complexity → Bug Fixes:**

- In the Semantic UI project, Max Complexity shows a strong correlation with Bug Fixes (0.8), suggesting that higher complexity leads to an increase in bug fixes. This aligns with our expectations that greater complexity results in more difficult-to-maintain components, leading to more defects.
- In the React Project, the correlation between Max Complexity and Bug Fixes is weaker (0.27), indicating that while complexity still impacts maintenance, its effect is not as significant as in Semantic UI. The modular architecture in React Icons helps mitigate complexity's effect on maintainability.

##### **LOC → Bug Fixes:**

- The Ant Project shows a moderate correlation (0.44) between LOC and Bug Fixes, suggesting that larger components tend to require more bug fixes.
- The Semantic UI project shows a very strong positive correlation (0.88) between LOC and Bug Fixes, reinforcing the idea that the size of the codebase is directly proportional to the maintenance burden.
- The React Project shows a weaker correlation (0.27), indicating that larger codebases in React may not be as strongly correlated with bug fixes.

#### 2. Mutual Relationships Based on Correlation Data

##### **Bug Fixes ↔ Refactors:**

- In the Ant Project, there is a very high correlation (0.91) between Bug Fixes and Refactors, suggesting a strong mutual influence. The more bugs are fixed, the more often the code is refactored, and vice versa.
- In the Semantic UI project, there is also a strong correlation (0.82) between Bug Fixes and Refactors, highlighting how bug fixes lead to necessary code improvements and refactoring, which, in turn, prevent future bugs.

##### **Max Nesting Depth ↔ Refactors:**

- In the Ant Project, there is a moderate correlation (0.45) between Max Nesting Depth and Refactors, suggesting that deeply nested components might require more frequent refactoring.
- In the React Project, the correlation between Max Nesting Depth and Refactors is weak (0.11), suggesting that nesting depth doesn't significantly influence the need for refactors in this project.
- Semantic UI shows a moderate correlation (0.6) between Max Nesting Depth and Refactors, indicating that deeper nesting could lead to more frequent refactoring, though it's not as strong as in Ant Project.

### **3. Parallel Factors Based on Correlation Data**

#### **Max Complexity and Max Nesting Depth:**

- Semantic UI shows a strong correlation (0.79) between Max Complexity and Nesting Depth, suggesting that more complex components tend to have deeper nesting, which may add to the difficulty of maintenance.
- The Ant Project shows a lower correlation (0.43), indicating that while there is some connection between complexity and nesting depth, they are not as strongly linked in this case.
- In the React Project, there is a relatively weak correlation (0.27) between Max Complexity and Max Nesting Depth, suggesting that while these factors are related, they don't necessarily influence each other in a straightforward manner.

#### **Merges → Maintenance:**

- All three projects show a strong positive correlation between Merges and maintenance activities (such as Bug Fixes, Refactors, Updates), indicating that more frequent code merges are associated with higher maintenance activity.
- Semantic UI shows the highest correlation (0.91-0.97) with Merges, indicating that frequent merging of code changes demands greater maintenance efforts to ensure code quality.
- React Project shows a moderate correlation (0.86), suggesting that merges require maintenance, but the relationship is not as strong as in Semantic UI.
- The Ant Project shows a very strong correlation (0.98) between Merges and Bug Fixes, highlighting the significant role merges play in ongoing maintenance.

## Summary

The analysis of the correlation heatmaps reveals several key insights about the relationship between various metrics and maintenance activities across the three projects. Specifically, **Max Complexity** and **LOC** strongly correlate with **Bug Fixes** in certain projects, highlighting the importance of managing complexity and code size for long-term maintainability. Additionally, mutual relationships between **Bug Fixes** and **Refactors** indicate the interconnected nature of code maintenance, where regular bug fixes often lead to necessary code improvements and refactoring. Finally, **Merges** consistently correlate with maintenance activities, underscoring the role of continuous integration in ensuring code stability and preventing defects.

## 5.2 Regression Analysis

To provide an additional layer of understanding regarding how component size, complexity, and nesting depth influence Bug Density, we conducted a multiple linear regression analysis for each project. This method differs from correlation analysis by isolating the specific contribution of each metric (e.g., Lines of Code (LOC), Max Complexity, Max Nesting Depth, Refactoring Frequency, Commit Frequency, Code Churn) to Bug Density, while controlling for the influence of other variables. This approach reveals which factors most significantly affect bug occurrence.

### 5.2.1 Explanation of Key Regression Terms

- **Coefficients:** In a regression model, coefficients indicate how much a change in a predictor variable, such as Lines of Code (LOC) or Max Complexity, affects the response variable, which in this analysis is Bug Density. A positive coefficient indicates that increasing the predictor variable is connected with an increase in bug density, whereas a negative coefficient indicates a potential decrease in defect rates.
- **Constant (Const):** The constant term, often known as the intercept, indicates the baseline value of Bug Density when all predictor factors (such as LOC and Complexity) are zero. This value serves as a baseline for the model's projections, indicating the expected degree of faults in the absence of the measured influences. It reflects the underlying defect rate, which may be influenced by unmeasured variables like project setup or coding standards.
- **Standard Error:** The standard error indicates the level of precision in the coefficient values. A lower standard error suggests a more accurate and dependable estimate, implying that the predictor variable's relationship with Bug Density is measured consistently. In contrast, a bigger standard error indicates greater uncertainty in the estimate.

- **T-Values and P-Values:**The t-value measures the strength of the relationship between a predictor variable and bug density.Higher t-values suggest a stronger relationship. The p-value denotes the statistical significance of this association; a p-value less than 0.05 shows that the effect of the predictor variable on Bug Density is statistically significant and unlikely to have occurred by chance.
- **Confidence Intervals (95% CI):** The 95% Confidence Interval provides a range of values that likely contains the true value of each coefficient. It's like saying, "We're 95% sure that the real effect of this metric on Bug Density falls somewhere between these two numbers." If the interval is narrow, it means we have a precise estimate of the coefficient's impact. But if the interval is wide, there is more uncertainty about its true effect. Importantly, if the interval includes zero, it suggests that the metric might not have a meaningful influence on Bug Density, since the true effect could be zero or even nonexistent.

## 5.2.2 Project-Specific Results

**Figure 5.7: Ant Project Regression Results**

Variable	Coefficient	Std. Error	t-value	P-value	95% CI	
					Lower	Upper
const	0.21	0.01	22.27	0.00	0.20	0.23
LOC	-0.00	0.00	-13.76	0.00	-0.00	-0.00
Max Complexity	-0.02	0.00	-7.23	0.00	-0.02	-0.01
Max Nesting Depth	-0.02	0.02	-0.83	0.41	-0.05	0.02
Refactoring Frequency	-0.01	0.01	-2.89	0.00	-0.02	-0.01
Commit Frequency	-0.00	0.00	-2.50	0.01	-0.01	-0.00
Code Churn	0.01	0.00	4.20	0.00	0.01	0.02

**Table 5.1. Regression results for Ant Project**

**Findings:** Ant Project's results reveal statistically significant negative coefficients for LOC and Max Complexity, suggesting that larger or more complex components show reduced Bug Density. This unusual finding may be due to Ant's emphasis on modular design and continuous refactoring, which appears to effectively control defects even in complex components.

**Figure 5.8: React Icons Project Regression Results**

**Findings:** In React Project, Max Complexity shows a positive and statistically significant relationship with Bug Density, indicating that higher complexity correlates with more

Variable	Coefficient	Std. Error	t-value	P-value	95% CI	
					Lower	Upper
const	0.123	0.053	2.342	0.039	0.007	0.239
LOC	0.000	0.001	-0.268	0.794	-0.001	0.001
Max Complexity	0.000	0.000	2.303	0.042	0.000	0.000
Max Nesting Depth	0.000	0.000	-0.405	0.693	0.000	0.000
Refactoring Frequency	-0.058	0.068	-0.853	0.412	-0.207	0.091
Commit Frequency	0.000	0.018	0.047	0.963	-0.039	0.041
Code Churn	-0.002	0.035	-0.050	0.961	-0.079	0.075

**Table 5.2.** Regression results for React Icons Project

defects. This aligns with traditional expectations that greater complexity increases the likelihood of bugs. However, the low R-squared value suggests other factors not included in this analysis may also impact bug occurrence.

### Figure 5.9: Semantic Project Regression Results

Variable	Coefficient	Std. Error	t-value	P-value	95% CI	
					Lower	Upper
const	0.152	0.019	8.121	0.000	0.115	0.189
LOC	0.000	0.000	-2.354	0.020	-0.001	0.000
Max Complexity	-0.004	0.007	-0.544	0.587	-0.019	0.011
Max Nesting Depth	-0.037	0.032	-1.155	0.249	-0.100	0.026
Refactoring Frequency	0.000	0.008	-0.045	0.964	-0.017	0.016
Commit Frequency	-0.003	0.005	-0.651	0.516	-0.012	0.006
Code Churn	0.0068	0.007	1.018	0.310	-0.006	0.020

**Table 5.3.** Regression results for Semantic Project

**Findings:** For Semantic UI, LOC has a statistically significant negative relationship with Bug Density, indicating that larger, modular components tend to have fewer defects. This may reflect the impact of modular structure on maintainability, supporting the idea that larger components can be stable if organized with clear boundaries.

### 5.2.3 Cross-Project Insights from Regression

The regression analysis across all projects reinforces that:

1. **Continuous Refactoring:** In Ant Design, regression results showed a negative coefficient for refactoring frequency, indicating that components with more frequent

refactoring activities tended to have lower bug densities. Additionally, the correlation heatmap (Figure 5.4) suggests a strong association between refactoring and bug fixes, implying that regular updates help prevent defect accumulation by maintaining cleaner and more manageable code structures. This connection highlights the importance of systematic refactoring in reducing bug density over time.

2. **Modular Design:** In React and Semantic UI, modular architecture appears to mitigate the risks of high complexity and size.
3. **Complexity Management:** Managing complexity at the component level, as seen in React, reduces the likelihood of defects, even within larger projects.

### 5.3 Summary of Findings

This chapter's analysis provides a comprehensive view of how component characteristics affect maintainability in React projects. Four principal findings emerged, each shedding light on the interplay between component metrics and bug density:

- **Component Size and Bug Density:** For projects like Ant Design and Semantic UI, larger components that are well-modularized do not exhibit notably higher bug densities. This indicates that component size alone does not predict defect rates; instead, effective modularization and well-considered structure are key factors in sustaining maintainability.
- **Complexity and Bug Density:** Complexity has a significant impact on maintainability. In the case of React Icons, higher complexity levels consistently correlate with increased bug density, underscoring the importance of complexity management to reduce defects. Conversely, Ant Design and Semantic UI show weaker correlations, suggesting that these projects have effective practices for managing complexity, which in turn supports code quality.
- **Nesting Depth:** Across all examined projects, maximum nesting depth shows only a weak correlation with bug density. This finding implies that React's component-based design helps mitigate the potential drawbacks of deep nesting, ensuring it doesn't heavily impact maintainability. However, excessive nesting can still add structural complexity, potentially complicating debugging efforts.
- **Maintenance Practices:** Ant Design illustrates the benefits of ongoing maintenance activities, such as regular refactoring and updates. Components that undergo frequent bug fixes, feature updates, and refactoring efforts tend to maintain lower bug densities, demonstrating that proactive maintenance practices play a crucial role in preserving code quality and reducing defects over time.

Together, these findings emphasize the importance of a balanced approach to component size, complexity, and maintenance. For React applications, incorporating modular struc-

ture, managing complexity, and prioritizing active maintenance can significantly enhance long-term stability and maintainability.

## 5.4 Developer Recommendations Based on Findings

Based on the analyses of Ant Design, React Icons, and Semantic UI, the following recommendations offer developers specific strategies for maintaining and improving React applications, particularly in large-scale and complex projects. Each recommendation is directly informed by specific findings from the data.

### Emphasize Modularity with Focused Refactoring

- **Insight from Ant Design:** The scatter plot of Lines of Code (LOC) vs. Bug Density (Figure 5.1) reveals that larger components do not necessarily have a higher bug density if they are modularized. Furthermore, in the Ant Design correlation heatmap (Figure 5.4), there is a moderate correlation between LOC and bug fixes, which suggests that modular boundaries can isolate defects and reduce bug frequency even in large-scale projects.
- **Recommendation:** Adopt modularization practices across projects of any scale, such as creating shared utility components and minimizing code redundancy. Modularity reduces defect rates by isolating changes and supporting code growth without introducing technical debt. This recommendation stems from the correlation observed in Ant Design's structure, where modularization practices appear to help manage bug density effectively, even for large components.

### Proactive Complexity Management

- **Insight from React Icons:** The scatter plot of Max Complexity vs. Bug Density (Figure 5.2) shows a notable correlation between high cyclomatic complexity and increased bug density in React Icons. This relationship is further confirmed in the React Icons correlation heatmap, (Figure 5.5) where Max Complexity has a significant positive association with bug fixes. These findings highlight that even smaller projects benefit from managing complexity proactively.
- **Recommendation:** Implement complexity management tools (e.g., ESLint, SonarQube) across projects to set thresholds for cyclomatic complexity. This is particularly beneficial in smaller projects, where managing complexity from the outset can prevent future technical debt. The recommendation arises from the observed trend that higher complexity correlates with increased defect rates, especially in projects like React Icons.

### Frequent Refactoring for Large and Critical Components

- **Insight from Ant Design:** In Ant Design, the correlation heatmap (Figure 5.4) shows

strong correlations between bug fixes, refactoring, and feature additions. This pattern indicates that continuous refactoring is associated with lower bug density in high-maintenance components. Frequent updates appear to support stability, as illustrated in the scatter plots of complexity and bug density for Ant Design.

- **Recommendation:** Develop a refactoring schedule for high-maintenance components, targeting those with high update frequency or bug density. This recommendation is based on observed patterns where regular updates in Ant Design appeared to support stability in large and complex components.

### **Limit Deep Nesting for Clarity and Stability**

- **Insight from Semantic UI:** The scatter plot of Max Nesting Depth vs. Bug Density (Figure 5.3) indicates a pattern where deeply nested components show a slight increase in bug density in Semantic UI. Additionally, the Semantic UI correlation heatmap (Figure 5.6) demonstrates a moderate correlation between nesting depth and bug fixes, suggesting that deep component nesting may add structural complexity, complicating debugging and impacting stability.
- **Recommendation:** Set maximum nesting depth limits, particularly in medium and large projects, to avoid maintenance issues. Shallow nesting enhances modularity and ease of understanding, as the findings from Semantic UI indicated, here limiting nesting depth correlated with improved maintainability.

### **Continuous Code Churn and Incremental Improvement**

- **Observation Across Projects:** Across all projects, scatter plots and correlation heatmaps indicate that regular updates (code churn) positively impact bug density when paired with systematic refactoring. In projects with frequent minor updates, as observed in Ant Design, lower bug densities are maintained over time.
- **Recommendation:** Encourage a culture of continuous improvement by making incremental changes to enhance code clarity, remove redundancy, and detect defects early. This recommendation is based on the overall analysis, where frequent updates and refactoring were associated with improved maintainability.

## 6. CONCLUSION

This thesis explored the impact of component characteristics—specifically size, complexity, and nesting depth—on the maintainability of React applications. By analyzing three distinct open-source projects (Ant Design, Semantic UI, and React Icons), the study established key relationships between component structure and defect rates, represented by bug density.

The analysis revealed that modularity and complexity management are essential factors in maintainable React design. For large components, a well-modularized structure appears to reduce bug density, indicating that modularity supports maintainability even in scaled codebases. Conversely, high cyclomatic complexity was consistently linked with increased bug density, especially in smaller projects without a robust modular structure, underscoring the importance of proactive complexity control.

Nesting depth was also shown to impact maintainability: projects with deeply nested structures tended to have higher bug densities, emphasizing the value of shallow nesting, particularly in larger codebases where readability and testability are crucial. Finally, continuous refactoring and incremental improvements demonstrated a strong correlation with lower bug densities across projects, reinforcing that an iterative approach to code maintenance can mitigate long-term technical debt.

Together, these insights emphasize that maintainability in React applications relies on a balance of modular design, complexity control, and active maintenance practices. By incorporating modular structure, managing complexity, and prioritizing active maintenance, developers can ensure long-term stability and reduced defects in their React projects.

Despite the contributions of this research, there are certain limitations to consider. The study is based on large-scale open-source React projects, which may not fully reflect the diversity of React applications in different settings, especially in commercial or enterprise environments. Additionally, while this research focused on component size, complexity, and nesting depth, future studies could explore the impact of testing practices, developer activity, and code review processes on maintainability. These factors could further refine our understanding of how maintainability is sustained over time.

For future work, it would be beneficial to extend the analysis to other JavaScript frameworks, such as Angular or Vue, to determine whether similar patterns exist across different

development ecosystems. Furthermore, examining the influence of collaborative development practices and automated testing could provide further insights into improving the maintainability of React applications and other frameworks. Exploring these additional factors would offer a more holistic view of how maintainability can be achieved in large-scale projects.

## **Limitations**

**Consideration of Additional Metrics:** This research initially explored the possibility of incorporating additional metrics related to the size of bug-fixing changes, such as lines of code added or deleted and the number of files modified. However, these metrics were ultimately excluded from the final analysis. The preliminary exploration indicated that they did not provide significant new insights beyond what was already captured by the existing Bug Density metric. Furthermore, variability in commit practices, such as developers grouping changes into single commits or splitting them into multiple commits, introduced noise that could obscure meaningful patterns related to component size and complexity. Future research could examine the impact of change size on maintainability in projects with more standardized commit practices, potentially revealing deeper insights into the relationship between change size and maintainability.

## REFERENCES

- Baggen, R., Correia, J., Schill, K., & Visser, J. (2012). Standardized code quality benchmarking for improving software maintainability. *Conference Proceedings*.
- Behnamghader, P., & Boehm, B. (2019). Towards better understanding of software maintainability evolution. *Systems Engineering in Context*, 593–603.
- Boehm, B., & Papaccio, P. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10), 1462–1477.
- Bogner, J., & Merkel, M. (2022). To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. *arXiv preprint arXiv:2203.11115*.
- Bogner, J., Wagner, S., & Zimmermann, A. (2017). Automatically measuring the maintainability of service-and microservice-based systems. *Journal Name*.
- Bytes, R. (2021). Best practices for optimizing performance in react.js applications.
- Chen, J.-C., & Huang, S.-J. (2009). An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*.
- Dinku, Z. (2022). React.js vs. next.js.
- Editorial, I. (2024). Optimizing frontend performance with react and javascript: Techniques and best practices. *International Journal of Novel Research and Development*, 189–195.
- Elmidaoui, S., Cheikhi, L., & Idri, A. (2019). Empirical studies on software product maintainability prediction: A systematic mapping and review. *Journal Name*.
- ESLint. (2016). Eslint: Pluggable javascript linter.
- Facebook. (2023). Thinking in react [React documentation].
- Fedosejev, A. (2015). React.js essentials. *Conference Proceedings*.
- Han, J. (1997). Designing for increased software maintainability. *Journal Name*.
- Khuat, T. (2018). Developing a frontend application using reactjs and redux. *Conference Proceedings*.
- Kuznetsov, A., & Kuznetsova, E. (2022). Comparison of redux and react hooks methods in terms of performance. *CEUR Workshop Proceedings*.
- Misra, S. (2005). Modeling design/coding factors that drive maintainability of software systems. *Journal Name*.
- Pavić, F., & Brkić, L. (2021). Methods of improving and optimizing react web-applications. *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, Opatija, Croatia.

- Phan, T. (2019). Software maintainability in web development for a periodical sporting event: Case: Sell games 2020 hosted by lahti. *Journal Name*.
- Poudel, J. (2023). Library management system with react.js. *Conference Proceedings*.
- Redux. (2021). A predictable state container for javascript apps.
- Rosene, A., & Connolly, J. (1981). Software maintainability: What it means and how to achieve it. *IEEE Transactions on Software Engineering*.
- Singh, M., & Srinivasan, V. (2024). React hooks: A paradigm shift in state management and side effects. *International Journal of Scientific Research and Engineering Development*, 7(4), 847–854.
- Tran, K. (2023). State management in react.
- Viljanen, J. (2015). *Measuring software maintainability: A comparative analysis of techniques* [Master's thesis, Aalto University].