User-based activity logging and analysis to improve system maintenance

C Scheepers

orcid.org/ 0000-0001-6089-7110

Dissertation accepted in fulfilment of the requirements for the degree *Master of Engineering in Computer and Electronic Engineering* at the North-West University

Supervisor: Dr J Prinsloo

Co-Supervisor: Dr JC Vosloo

Abstract

Title:	Using user-based activity logging and analysis to prioritise software maintenance
Author:	Mr Cornelius Scheepers
Supervisor:	Dr Jaco Prinsloo
Degree:	Master of Engineering in Computer and Electronic Engineering
Keywords:	software maintenance, logging mechanism, user activities, system utilisation, maintenance prioritisation, web-based

Software maintenance is continuous and forms part of the software development process. Research suggests that 15% of the total software development cost should be allocated to implementing a maintenance model for a software system. Unused components or non-compliant software will increase over the project's life cycle. The discontinuation of some systems may reduce the resources required to maintain the system. Deciding how many resources to allocate for maintenance for each subsystem can be challenging without a suitable software maintenance prioritisation method.

Software logging is crucial to improve software maintenance by troubleshooting. In extensive software systems, logging enables the development team to monitor specific events. A suitable logging mechanism can identify the most used software system. In web-based applications, the user interactions with each software system can be determined by capturing user-based events.

Analysing logs can be challenging if the logging mechanism does not track the desired user-based events. Developing a method to track these events for a specific purpose is more efficient and reliable. The author suggests integrating the method to create a logging mechanism and a log analysis to prioritise software maintenance.

The method for this study was divided into two main functional parts, namely the logging mechanism and the log analysis. The characteristics of these user-based events, user types, and user-based attributes are defined. The logging mechanism captures any user activity on the software systems, and the logging points in strategic locations in the software systems capture the log attributes. HTTP requests have more significant and relevant data about a specific event. Additional data from request parameters are obtained as metadata that can be used for system diagnostic purposes.

For the log analysis for this study, log quality is monitored to ensure that event logs are consistent, reliable, and complete to create prioritisation recommendations for software maintenance. A test system validates the method's work, making modifications where needed. The results of this method applied to case studies proved that the method can prioritise software maintenance. The results are evaluated and discussed, and positive and negative points are highlighted by implementing this method. Therefore, the study objectives have addressed the need to develop a method for log analysis for software maintenance by creating a suitable logging mechanism to capture user-based event logs.

Acknowledgements

I want to thank my parents, Albert and Mirriam Scheepers, for their unwavering support through all these years. Your love and encouragement have been my guiding light; I couldn't have done it without you. I also want to thank my brothers, Tyron and Danzel, and my sister, Claudia, for always being with me on this journey.

I'm also profoundly grateful to ETA-Operations (Pty) Ltd for granting me the opportunity to pursue my master's degree through an IPGIP-bursary under the guidance of Prof. E.H. Mathews.

I want to extend my sincere appreciation to Dr. Jaco Prinsloo, my supervisor, and Dr. Jano de Meyer, my mentor, who helped me during my most difficult times. Your unwavering support and guidance were invaluable, and I am thankful for your mentorship.

A special thanks to my previous supervisors, Dr. Pieter Goosen and Dr. Jan Vosloo, for getting me started on my master's journey.

I would also like to express my gratitude to Megan Lowes for her meticulous proofreading of my work and her incredible patience with me. Your assistance was invaluable in the final stages of my journey.

Lastly, I want to thank my colleagues and friends for their support and encouragement throughout this journey. Your friendship has made this experience even more memorable and meaningful. A heartfelt thank you to Kaido, my beloved leopard gecko, who has been my unwavering and most significant source of support.

Table of contents

Abstract . . . . . **i**

Acknowledgements . . . . . **iii**

Table of contents . . . . . **iv**

List of figures . . . . . **vi**

List of tables . . . . . **vii**

Nomenclature . . . . . **ix**

1 Introduction **1**
- 1.1 Background 2
- 1.2 State of the art 6
- 1.3 Problem statement 38
- 1.4 Objectives of the study 39
- 1.5 Overview of the dissertation 40

2 Methodology **41**
- 2.1 Preamble 42
- 2.2 Development of the solution 44
- 2.3 Investigate 45
- 2.4 Design 60
- 2.5 Conclusion 71

3 Results **72**
- 3.1 Introduction 73
- 3.2 Implementation 73
- 3.3 Verification 80
- 3.4 Case studies 86
- 3.5 Conclusion 109

4 Conclusion **110**
- 4.1 Discussion 111

4.2 Recommendations 118
4.3 Conclusion 120

References **121**

A Logging practise in software engineering **127**

B Case study results **129**
B.1 Case Study A 129
B.2 Case Study B 135
B.3 Case Study C 138

List of figures

1.1 Project plan with included technical debt repayment phase 5
1.2 Resource cost of software maintenance 7
1.3 IEEE Standard 1219 model for software maintenance 11
1.4 Maintenance flow model 13
1.5 Quality model for event logs 17
1.6 An illustrative example of log parsing 22

2.1 Basic design of a software system 42
2.2 Logging mechanism basic system design 43
2.3 Development of solution 44
2.4 MVC architecture for most web-based applications 61
2.5 User-based activity log classification flow diagram 64
2.6 Server-side log parsing flow diagram 66
2.7 ERD of user activities 68
2.8 Database interaction flow diagram 68
2.9 Maintenance prioritisation flow diagram 70

3.1 JavaScript event propagation 76
3.2 HTML element capturing flow diagram 78
3.3 Logging point operation for test system 79
3.4 Interactive user activity viewer 80
3.5 JSON test request parameter data 81
3.6 Interactive user activity viewer 81
3.7 Interactive user activity log analysis 82
3.8 User activity types breakdown of Case Study A 88
3.9 System utilisation breakdown of Case Study A 88
3.10 User activity types breakdown of Case Study B 92
3.11 System utilisation breakdown of Case Study B 93
3.12 User activity types breakdown of Case Study C 98
3.13 System utilisation breakdown of Case Study C 99

A.1 The distribution of the papers' published years 128

List of tables

1.1 System Development Life Cycle Phases . . . 2
1.2 Software maintenance types . . . 8
1.3 Event logs usage . . . 14
1.4 Problems with too much logging . . . 18
1.5 Basic log event attributes . . . 20
1.6 Web analytic for user-based data . . . 23
1.7 Software maintenance state of the art sub-topics . . . 26
1.8 Event logging state of the art topics . . . 27
1.9 A log analysis using state of the art topics . . . 27
1.10 State of the art . . . 28

2.1 Functional requirements of the solution . . . 45
2.2 Log attributes functional requirements (F/R 1) . . . 46
2.3 Requirements for an event to be classified as a user-based activity . . . 47
2.4 User activity types . . . 48
2.5 Logging attributes . . . 49
2.6 Logging points functional requirements (F/R 2) . . . 52
2.7 Logging point requirements . . . 52
2.8 Log attributes for database storing of the event logs . . . 54
2.9 Log analysis functional requirements (F/R 3) . . . 54
2.10 Log quality functional requirements (F/R 3.1) . . . 55
2.11 Log analysis tool functional requirements (F/R 3.2) . . . 56
2.12 Maintenance prioritising functional requirements (F/R 4) . . . 57
2.13 System utilisation analysis categories functional requirements (F/R 4.1) . . . 58
2.14 Maintenance prioritisation factor functional requirements (F/R 4.2) . . . 59

3.1 Test user activity types . . . 74
3.2 Logging attributes . . . 74
3.3 Logging quality assessment of the test system . . . 83
3.4 Data for validating test system . . . 84
3.5 Test data . . . 85
3.6 Case studies . . . 86
3.7 Case Study A activity types . . . 87

3.8 Logging quality assessment of Case Study A 89
3.9 Case Study A's upper quartile maintenance performance 90
3.10 Logging quality assessment of Case Study B 94
3.11 Case Study B's upper quartile maintenance performance 95
3.12 Case Study A activity types 96
3.13 Logging quality assessment of Case Study C 100
3.14 Case Study C's upper quartile maintenance performance 101
3.15 Functional requirements addressed 106

4.1 State of the art 112
4.2 Study validation 114

A.1 G. Rong's inclusion selection criteria 127
A.2 G. Rong's exclusion selection criteria 128

B.1 Case Study A results 129
B.2 Case Study B results 135
B.3 Case Study C results 138

Nomenclature

Abbreviations

Acronyms and Their Full Forms

Acronym	Full Form
AJAX	Asynchronous JavaScript and XML
BI	Business Intelligence
CPU	Central Processing Unit
DOM	Document Object Model
ERD	Entity Relationship Diagram
F/R	Functional Requirements
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
LCP	Life-Cycle Phases
MVC	Model-View-Controller
OSS	Open-source software
PHP	Hypertext Preprocessor
SDLC	Software Development Life Cycle
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
JSON	JavaScript Object Notation

Chapter 1

Introduction

1.1 Background

Most businesses use digital products and services to maximise their profits [1]. This increases the need to create new, innovative software systems to meet the specific needs of users. Software projects can vary in size, type, and degree of difficulty of implementation. Therefore, the quality of software is crucial. There are defined attributes and characteristics that software systems need to adhere to, which are termed software quality [2].

A *System Development Life Cycle (SDLC)*[1] methodology, such as the Agile Software Development methodology, is crucial to making the software development process efficient and predictable. This enforces various degrees of discipline in the software development process to ensure that the quality of the software is acceptable for the user's requirements [2, 3]. Table 1.1 lists the Life-Cycle Phases (LCP) of the SDLCs.

Table 1.1: *System Development Life Cycle Phases [2]*

Life-Cycle phase	Description
Initiation	The sponsor identifies a need or an opportunity, creating a concept proposal for the new opportunity. This new opportunity will require a software solution to make the project successful for all stakeholders.
System Concept Development Phase	In this phase, the feasibility and suitability of the concept proposal are reviewed and need approval. The Systems Boundary Document identifies the scope and requires additional approval before implementing the planning phases.
Planning	The project management plan and other documents are created in the planning phase. These documents define the available budget, project resources, activities, schedules, tools, and reviews.

Continued on the next page

[1] **System Development Life Cycle (SDLC)** highlights specific procedures, practise s and guidelines for software development. Refer to the source: DOJ, "SDLC", DOJ, Available: `https://www.justice.gov/archive/jmd/irm/lifecycle/ch1.htm` (visited on 2023-07-25)

Table 1.1: *(continued from the previous page)*

Life-Cycle phase	Description
Requirements Analysis	In this phase, user requirements are defined and analysed. The functional requirements are created with the Functional Requirement Document. Non-functional requirements are also defined to ensure the software system operations will not deviate from working within the non-functional specifications. These non-functional requirements can be harmful and costly to the software system's operations and potentially reduce its usability or life cycle.
Design	In this stage, the detailed requirements are completed in the System Design Document, which describes the detailed logic specifications of the software system.
Development	In this phase, the design specifications are converted into an executable software system. This also includes acquiring other third-party or internal software to install in specific software environments, creating and testing databases, performing test readiness reviews, and other software development activities.
Integration and Test	In this phase, the software systems are integrated and systematically tested to examine whether they meet all functional requirements and other accreditation activities for the approval of the test and the user's approval using user tests.
Implementation	This phase is initiated after the software systems pass the testing phase and are accepted by the users. This phase contains the implementation of the software system in a production environment and the deployment of the production version of the software. This phase continues until the software systems are operational in a final production environment.

Continued on the next page

Table 1.1: *(continued from the previous page)*

Life-Cycle phase	Description
Operations and Maintenance	In this phase, the software system's performance is continuously monitored according to the defined user requirements. Any additional modifications after the initial software development are done here, and other tasks are needed to maintain the software system. Post-implementation and In-Process Reviews for the software system form part of the documentation for this LCP.
Disposition	This phase is for any disposition activities to terminate the software system. Essential data are also preserved to reactivate the software system in the future. Any other termination policies and activities must be defined and documented. This ensures that the termination of the software system is performed correctly and completed and that the correct data are permanently removed or preserved.

There are various adaptations of the LCP listed in Table 1.1 for different SDLC methodologies used in the software development industry [3]. The planning phases of the SDLC should be well-documented, and the software architecture should be well-structured and defined to make it easier to implement the LCP related to development and maintenance [4].

Each LCP ensures that the software development is correctly implemented if it is part of the adopted SDLC methodology. The SDLC methodology is implemented for the entire software system life cycle. There will be alterations to the SDLC methodology to meet any new design and development requirements.

In implementing the SDLC methodology, some design patterns may drift away from the initial software designs [5]. This can be due to unforeseen issues during the development phase due to time or other resource constraints. The initial design of the software system may also not be flexible enough for any newer modifications that can occur in the operational and maintenance phases.

Software development could provide workarounds or shortcuts that resolve the identified problems in both situations. These short-term benefits will not always translate to good long-term software quality due to prioritising functionality above good software design patterns. The decrease in software code quality is caused by technical debt if the software system is not implemented with suitable SDLC practise s [5, 6].

Technical debt can be defined as technical compromises that software engineers and developers will introduce to a software system for short-term goals that can increase the complexity and sustainability of the system in the long term [1, 7]. With the need to consistently deliver software systems for more innovative, complex, and larger software systems, the risk of introducing technical debt also increases [2, 5].

The operations and maintenance phase of the software system is where technical debt is usually resolved or reduced to some extent. More technical debt will increase the maintenance efforts that need to be implemented to offset the negative impact that short-term goals can potentially create. Figure 1.1 represents the technical debt repayment to the software system during its entire life cycle.

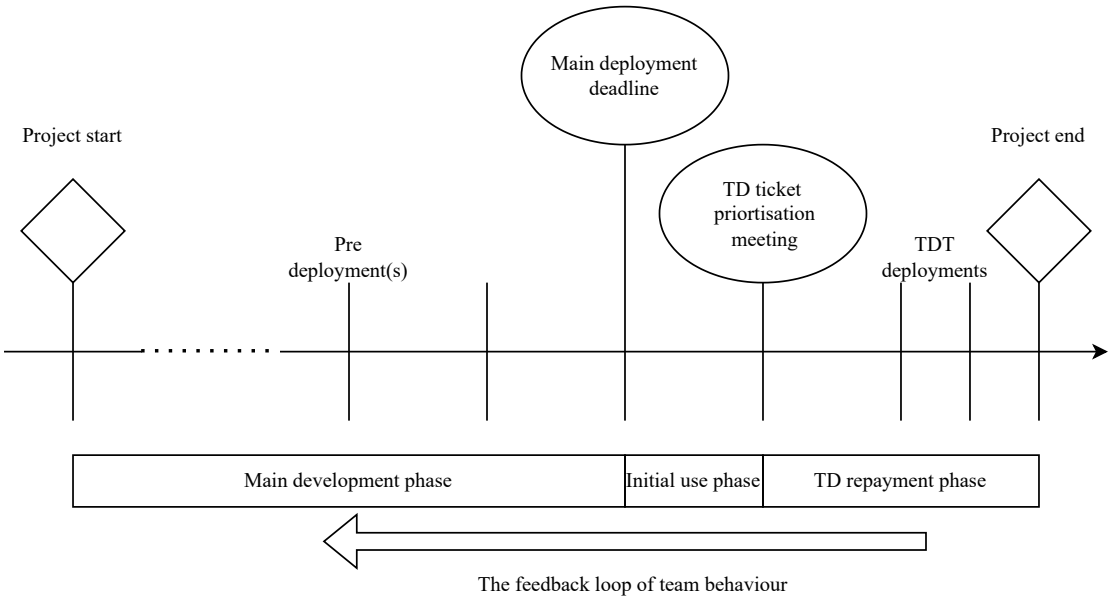

Figure 1.1: *Project plan with included technical debt repayment phase [8]*

Technical debt will always be present to some degree in software systems throughout their life cycle [8]. In Figure 1.1, technical debt repayment refers to software development activities that aim to resolve technical debt issues identified after the initial deployment of the software system.

Tickets are assigned to identified problems caused by technical debt when functional requirements are reviewed in the initial use phase. The technical debt tickets are then prioritised based on their importance and utilisation. This ensures that the development efforts in the Operations and Maintenance phase are efficiently implemented to ensure user satisfaction and sustainability of the software system.

According to the United States Department of Commerce, the software maintenance efforts of the SDLC Operations and Maintenance LCP in Table 1.1 will contribute approximately $60\% - 80\%$ of the total development cost for the entire software system life cycle [4, 9, 10]. Therefore, following good software maintenance practises is necessary to avoid [6]:

- additional software and hardware resources needed that can be costly,
- software quality issues,
- making any new modifications impossible without negatively impacting existing software features or systems and
- is shortening the usability of the software system, which can lead to earlier termination of the software system.

Maintenance of the LCP operation and maintenance software is essential in software development. It can directly reduce the cost and effort to create new software systems or modify them in the future and minimise technical debt [6, 11].

1.2 State of the art

1.2.1 Software maintenance

Maintenance of software systems is a continuous process and a reduced form of software development aimed at modifying software systems while preserving their integrity for current and future operations [4,12,13]. Software maintenance aims to improve the software in terms of the following:

- **correctness:** software systems always have some defects or faults that must be corrected to improve their traceability, consistency, and completeness.
- **enhancements:** existing software components need to be improved to adapt to changes in user requirements and improve system performance and sustainability.

A defined maintenance process must be followed to improve the -accuracy and enhancements of the software. According to the *IEEE Standard 1219*[2], software maintenance includes the following phases [14–16]:

- Identifying the problem or modification and classifying it.
- Analysing the identification of the maintenance issue.
- Designing the solution to implement maintenance.

[2] **IEEE Standards** documents are developed within the Technical Committees of the Institute of Electrical and Electronics Engineers (IEEE) Societies, and the Standards Coordinating Committees of the IEEE Standards Board [14].

- Implementing the solution.
- System testing of the modified software system.
- Acceptance testing on the fully integrated system.
- Meeting the delivery requirements of the modified software system.

These maintenance phases of software cannot be omitted when the software system is still active. Software maintenance must be implemented to ensure that the software system can keep up with the new user requirements defined in the future. This will increase the maintenance required on new and old systems [15, 17, 18]. Figure 1.2 represents the total cost of implementing software maintenance.

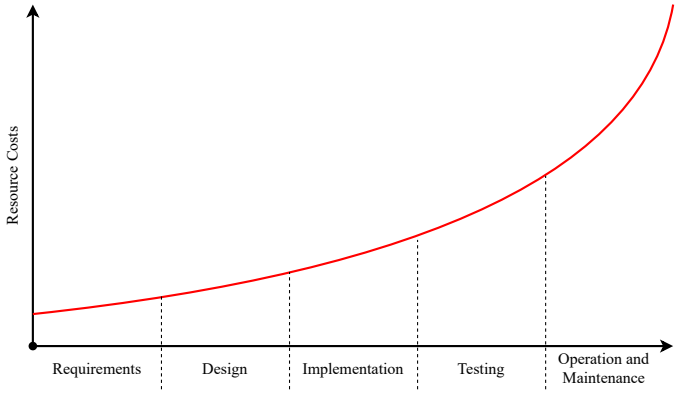

Figure 1.2: *Resource cost of software maintenance [9]*

In Figure 1.2, the total cost of the resources will increase significantly as the need for software maintenance increases. As previously stated, software maintenance can use up to $60\% - 80\%$ of the unlimited resources, and it can be expected that most of the resource costs will be for the Operations and Maintenance LCP. These software resources that cost both money and time can include the following:

- software developers and other support staff involved in the software maintenance process and
- software development tools and services such as testing software environments, analysis tools, online surveys, software fault reporting systems, etc.

Software maintenance types

Maintenance problems or modifications are regularly identified and addressed based on an initial priority ranking. This priority classification is determined using classification models to determine what type of maintenance is needed, as described in Table 1.2 for the Operations and Maintenance LCP of Table 1.1 [10, 19].

According to Table 1.2, adaptive and perfective maintenance types account for approximately

Table 1.2: *Software maintenance types [15, 19]*

Maintenance type	Description	% of maintenance activities
Adaptive	Adaptive maintenance in software systems is any modification or enhancement to keep it usable with a changing or changed software environment.	≈ 37.5%
Perfective	Perfective maintenance consists of modifications based on changing the end-user's new requirements. It can also improve the performance or maintainability of the software system in its life cycle.	≈ 37.5%
Corrective	Corrective maintenance is improvements made to fix certain defects or errors in a software system.	≈ 20%
Preventive	Preventive maintenance refers to improvements made to software systems that prevent problems in the future.	≈ 5%

75% of the total maintenance of software development for the Operations and Maintenance LCP. These maintenance types address technical debt issues that may arise after the initial software deployment. They are typically identified through technical debt tickets, as shown in Figure 1.1. Adaptive and perfective maintenance is critical to ensure that the software system continues to evolve and improve, meeting the system requirements to ensure that it is usable and feasible [20].

Software faults or defects that require repair and deployment are inevitable. These maintenance software changes are usually minor and aim to increase the accuracy of the software system. Preventive measures may also be taken to avoid technical debt issues in the future through preventive maintenance efforts.

However, prioritising available resources for certain parts of the software system to maintain and prevent or fix software defects can be challenging [14, 15]. The defect density of a software system is determined by the number of possible defects divided by the size of the software system, as shown in Equation (1.1) [21]:

$$Defect\ Density = \frac{CNDD}{KLoC}, \tag{1.1}$$

where:

- $CNDD$ is the cumulative number of defects in the post-release version of the software system

- $KLoC$ (thousands of lines of code) is the size of the observed executable code in a software system.

A lower defect density indicates good software quality [22,23]; however, it does not necessarily imply that the software meets all user requirements but that fewer possible faults exist.

In open-source software systems (OSS), the defect density tends to increase due to the size of the system and the number of developers working on it [21]. Adding more developers to improve a software system may not always lead to improvements in all maintenance types listed in Table 1.2.

This increase in the size and complexity of the software system may also lead to a higher defect density, increasing the need for corrective maintenance efforts. Therefore, it can further exacerbate developer challenges as they try to resolve maintenance issues [23].

Problems with implementing software maintenance

Under most circumstances, maintenance is implemented if a software system does not meet the required functions specified by the user or performance requirements [9,12]. Maintenance can be difficult to implement due to the following:

- **Problem domain being complex:** The software may not be well-defined or structured during the planning phases of LCP of Table 1.1. This is due to the size of software systems throughout their entire life cycle or the duplicate software components.

  There is a poor understanding of the system architecture or insufficient documentation about the software system when analysing maintenance problems [18]. Software engineers and developers tend not to create or update documentation as it is time-consuming when software needs to be delivered on schedule.

- **Difficulties of managing the development process:** Most companies will strive to increase their digital products and services throughout the life cycle of the software project to maximise possible profits with the invested resources [17]. Expanding the development process's production will only strain software systems' maintenance efforts [12].

  Software engineers and developers feel pressured to deliver software features on time [18, 24]. They will quickly feel overwhelmed and suffer from development burnout if the development process is not correctly managed to include additional software development by implementing maintenance.

- **Flexibility of the software:** Trying to predict the future architecture and modify it while preserving the integrity of the software may be difficult in software maintenance [25]. Software is considered flexible when it can be modified by modifying the problem

domain [9].

Most development teams will follow a software development methodology to create a future architecture that is modular and structured to preserve the development integrity of new software [26]. This will also have an impact on the type of maintenance activity (such as in Table 1.2) the development team will need to implement [7, 11].

- **Change in user's requirements:** In software development, users will often request additional requirements for the software systems that are delivered to them [9]. Modifying software systems may include new features and other features that change the initial architecture of the system. Maintenance of these systems is crucial to ensure that existing components of the system will work as intended with the new members that are added.
- **Environmental changes:** Rapid changes in software development are always present. As such, the need for more innovative solutions to solve new complex problems exists. These changes are not always compatible with existing software systems, even if the initial software architecture is well-defined [9].

  There will always be a need to improve the software system through third-party software updates and services. The software system must be modified to accommodate these changes, as it benefits its sustainability and operation.
- **Bad design:** Maintenance can be complex if the system is poorly designed when SDLC is implemented. Maintaining the system without making significant changes unfeasible. The complexity of the software system may be too complicated for some developers; this will lead to complexity in software systems within the mentioned problem domain [24].

Software maintenance prioritisation

Planning maintenance operations is challenging for most software engineers and developers due to time constraints and available resources [6]. The increased cost of software maintenance resources is usually due to a lack of planning or preventive measures to keep the software system from degrading [23]. Continuous analysis of identified maintenance problems or modifications enables software engineers and developers to create a preliminary plan to address these problems efficiently [13]. Implementing a suitable maintenance framework to resolve the issues identified reduces technical debt.

Various maintenance models can be used to solve these issues when implementing any of the maintenance software types. A software maintenance model is an abstract representation of the evolution of software systems to keep track of all the maintenance activities when implementing software maintenance [27]. The *IEEE Standard 1219* for software maintenance

is the standard that should be followed when planning software maintenance, as seen in Figure 1.3.

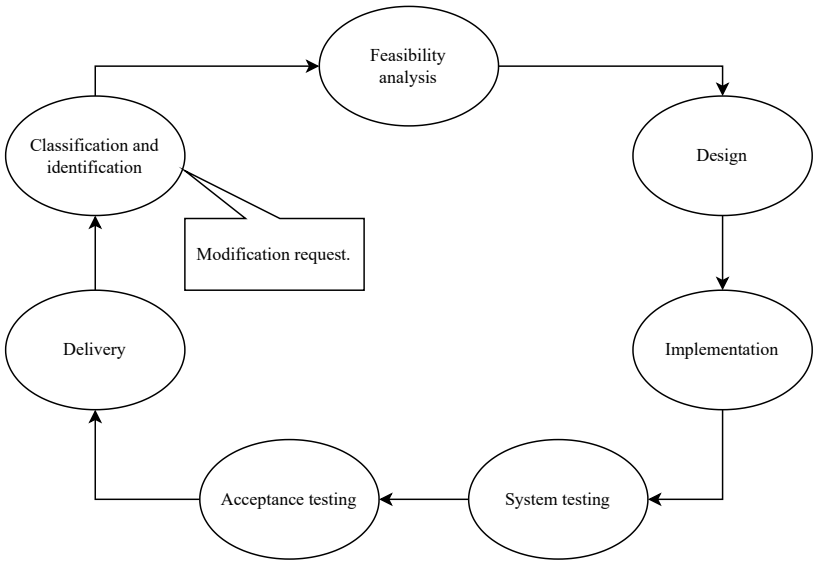

Figure 1.3: *IEEE Standard 1219 model for software maintenance [27]*

The software maintenance model in Figure 1.3 emphasises that software defects or faults must be identified and classified for maintenance. If the maintenance effort is worth implementing, a feasibility analysis of any required changes should be done. Several resources will be allocated to execute maintenance, affecting the design and implementation phase.

For feasibility analysis, using a system characterisation report may help identify possible maintenance focus points in a software system [28]. This will increase the effectiveness of designing solutions to implement maintenance because a system assessment focus can be made. The metrics can be defined as:

- positively or negatively impacts the performance of the system,
- fulfil the defined user's requirements,
- increase user engagement with the defect and fault fixes or expand existing software components with additional features and
- is worth a large portion of the user base to implement.

User engagement with the software system will determine whether the software system is sustainable in generating revenue for the organisation. It may be the most crucial focus metric for feasibility analysis to determine whether a maintenance effort is worth implementing [28]. Compared to the model in Figure 1.4, a maintenance flow model can resolve this problem.

Figure 1.4 is an example of a practical maintenance flow model that an organisation would likely use to implement software maintenance. Initially, a developer will complete a request

form that indicates a new problem or feature request that needs to be implemented [10].

After all maintenance tasks are defined, the development team will prioritise the higher-rated issues that need to be solved. This process will repeat until all the work for that specific software system is completed.

To fully follow the *IEEE Standard 1219* of implementing maintenance on a software system, the defects or areas of improvement should be identified. The analysis of event logs can detect hidden flaws or performance problems in a software system to implement software maintenance [29–31].

System and acceptance tests are essential to ensure the system is fully functional and meets user requirements. After the system is thoroughly tested and approved, it will be available to the user, and the maintenance process will start again when there are new improvements to the software system.

In Section 1.2.1, it was identified that software maintenance is essential to fulfilling the user's requirements. For any maintenance model to be effective, as described in Section 1.2.1, the software maintenance model must prioritise maintenance issues efficiently.

Most organisations' maintenance models will be based on Figure 1.3 to manage their Operation and Maintenance LCP maintenance efforts. Due to the problems with the implementation of maintenance discussed in Section 1.2.1 up to 50% of a software engineer or developer's total time invested in implementing maintenance is to understand what the software system is supposed to do [10].

If the issue is a problem in the software system, the severity of the problem must be assessed to decide on the priority level to resolve it. This type of maintenance is mainly corrective and can also be preventive if it is a possible solution to prevent software failures in the future [10]. Other maintenance requests are adaptive or perfective and are usually placed in the development team's task queue.

Prioritising maintenance for the user's requirement to extend usability and increase user satisfaction is preferable to maximise profits. When no suitable software maintenance model is used, software maintenance is often reactive [28]. However, this is not a sustainable maintenance policy for larger, more complex software systems.

A systematic characterisation of the software system needs to be made to prioritise maintenance more efficiently. In Section 1.2.1, user engagement with the software system has been identified as an essential metric when implementing maintenance.

Knowing what the user uses or interacts with in the software system provides valuable data to the development team. But to access those data, some form of tracking is needed to obtain the data automatically.

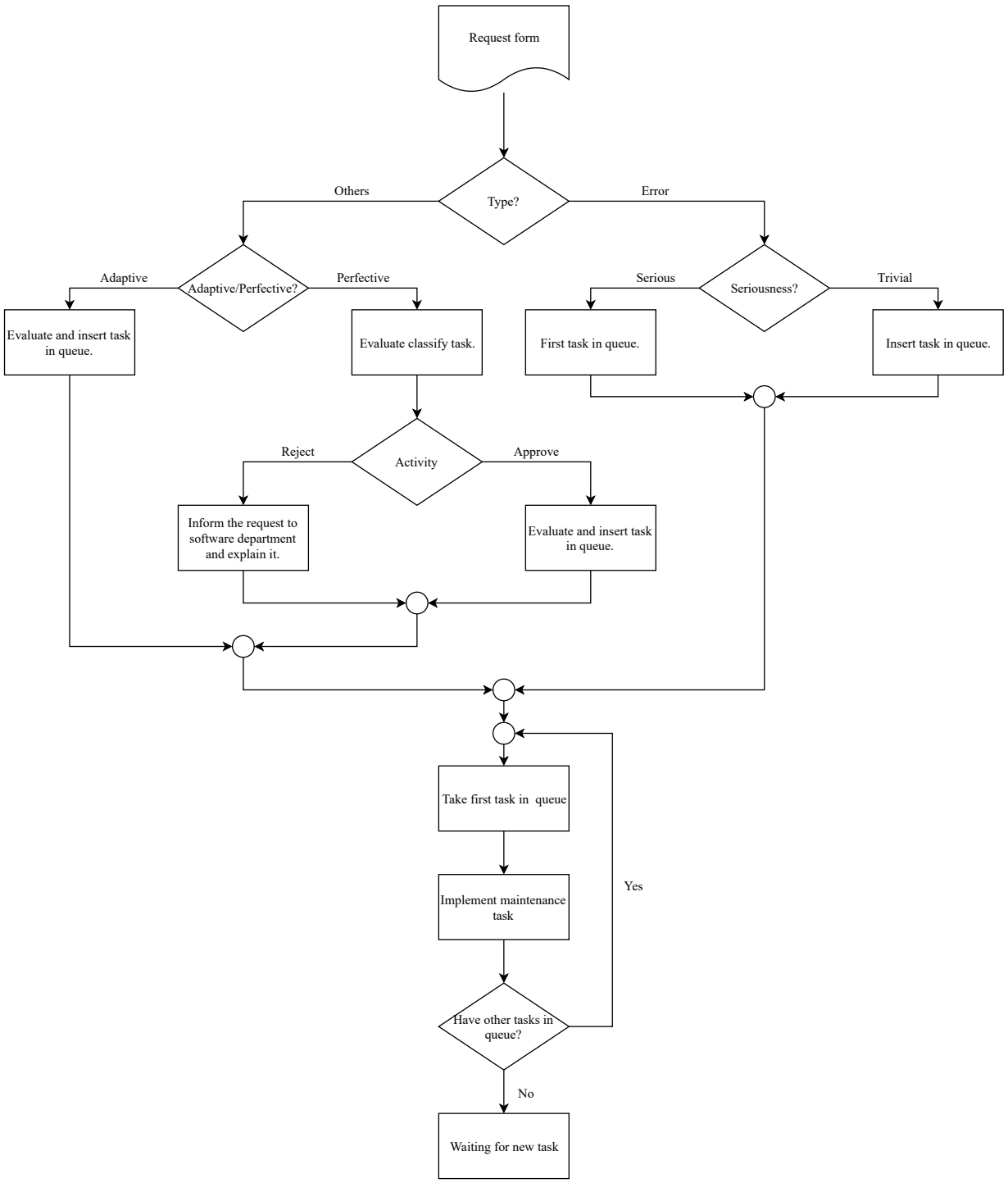

Figure 1.4: *Maintenance flow model [10]*

1.2.2 Event logging

As described in Section 1.2.1, a tracking method is needed to capture data on user engagement with the software system for maintenance purposes. It is a common practise in the software industry to record complex system run-time information in event logs. Developers or engineers can analyse these event logs later to solve software-related problems [32].

Event logging is a proven implementation to obtain information on the behaviour of software systems [33]. Event logs are textual files generated by the software system that collect data on reported events of interest during various software system operations [29, 33].

The technique to collect numeric or textual data that describes the behaviour of a computer system is called the event log [33, 34]. Event logs collect textual data containing records of events in a software system and are used for system management tasks, as seen in Table 1.3 [30, 33, 35]. Event logging has three main purposes [33, 34]:

- **state dump** reports the values of certain variables or data structures inside the software system.
- **execution tracing** is the reporting of certain states of the software system or what is currently happening in the software system and
- **event reporting** focuses on any desired events in the software system that have textual information of that event.

Event logs are used mainly for event reporting to support debugging and system integration activities to reduce the amount of code that needs to be inspected [33]. Table 1.3 is the industry's most common use of event logging.

Table 1.3: *Event logs usage*

Usage	Description
Debugging of software systems and services	Event logging is used mainly to record events or behaviours of software systems or services during its runtime [30].
Anomaly detection	Event logs can detect any abnormal system behaviour using an anomalous detection algorithm using log data [36]. This can also be used to find vulnerabilities or predict software environment defects [37].

Continued on the next page

Table 1.3: *(continued from the previous page)*

Usage	Description
Performance diagnosis	Software performance is important to produce quality software for the end user [33, 38]. Making informed decisions about improving the software system or service to improve performance and other financial and resource implications is also important. This type of performance event log of software systems or services is used to monitor the software system, which is useful for tuning resources, balancing load and checking system scalability throughout the life cycle of the software system or service [39].
Auditing	In a software environment, significant changes in database data may need to be recorded for audit purposes [30]. All establishments and enterprises must ensure compliance with industry regulations is met with their software systems by adding audit logs. They are also legally bound to have audit logs to provide evidence for any legal investigation or administrative tasks to maintain accountability.
Error and failure analysis	Event logs are used to analyse the failure behaviours of software systems, which allow software engineers or developers to understand system failure, find the root cause of these failures, prevent them, and improve the reliability of future versions of the system [29].
Analysis of security alerts	Security is a significant concern in any software environment or information technology infrastructure [37, 40]. It is essential to know the overall security status of the software system.

Logging practise in software development

Logging practise in software development is not always well documented, and multiple logging mechanisms can be implemented in the same software system [34, 41]. In modern software systems, logging practise is crucial to software development and maintenance throughout its life cycle [35].

In Appendix A, new studies have focused on providing practical logging practise guidelines to software engineers and developers. Logging in the industry uses many third-party log libraries and frameworks, such as Apache's log4net and Microsoft's ULS frameworks [35, 42].

Software engineers and developers can use these tools to implement logging in a compatible software system. However, it will still need to know how to place the logging points to obtain the desired logs strategically. The logging practise can be implemented using the guides provided by the tools and other online guides. When using a third-party logging mechanism, the software engineers and developers will, in most cases, need to:

- add the logging points in the software system at locations where it can capture the desired logs and
- enables the log parsing stage to write a log entry into a database.

Logging guides can give examples and suggestions on where to place the log points. However, it can still be difficult for software engineers and developers to identify these desired locations because logging guides are either mostly application-specific or the logging mechanism can only capture certain event types to create event logs.

For more custom logging, the software engineers and developers will need to develop new logging mechanisms. This introduces new requirements for the logging mechanism to be functional. For a log-keeping practise to be successful, two critical problems need to be resolved [32, 35, 42]:

- **What needs to be logged?**

  In Table 1.3, recording is diverse and will impact how the logging mechanism will be designed.

- **Where to log?**

  Different types of logs will only be present in the software system at certain locations during runtime. Knowing what to log narrows down which locations can be used to obtain certain events while they are present.

To answer these two questions, the event log must meet specific log quality requirements. This ensures the created logs are correct and consistent when extracted and viewed.

Logging quality

Software engineers and developers must make informed logging decisions for event logging. These decisions may negatively affect the software system's run-time operations and event logging efficiency [32, 42, 43]. An event log quality model is defined in Figure 1.5 to ensure that the event logs have consistent integrity when capturing the event log data.

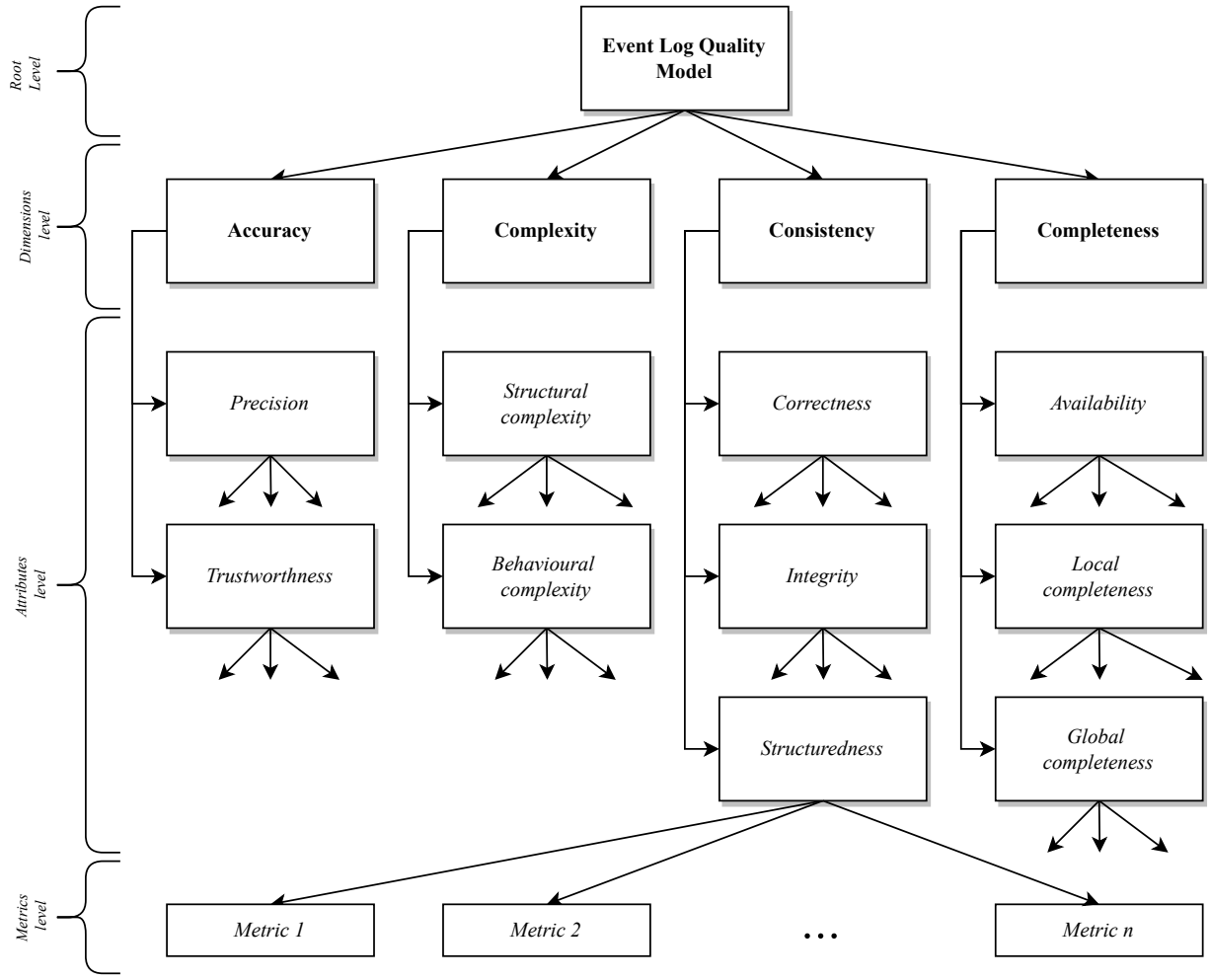

Figure 1.5: *Quality model for event logs [43]*

The event log quality model in Figure 1.5 comprises four different levels to define each property of the event log quality model:

- Root level, these are the main requirements for the event log quality model,
- Dimension level, which compromises four main dimensions of the event log quality model according to [43]: complexity, accuracy, consistency, and completeness of the event log,
- Attributes level, for which a set of quality attributes for the dimension level are defined,
- Measurement level, which should be defined in the design process of the logging mechanism to achieve the attributes and dimension levels.

Each of the four dimensions is discussed in detail in the Figure 1.5:

- **Accuracy of event logging**

  It can become difficult to record every event in a software system as these systems will become larger and more complex during their development life cycle [16]. Precisely capturing certain events that need to be logged must be consistent to ensure that the data are trustworthy and reliable. The log attributes of the event log also need to be precisely captured for each event log and should be correct.

  Capturing more event logs does not guarantee that the accuracy of the event log is acceptable, as the log's attributes can be incorrect or cause duplicated event logs if the data are the same in a sequence of event logs. Table 1.4 indicates common problems associated with the accuracy.

Table 1.4: *Problems with too much logging [42]*

Problem	Description
Excess code	Adding multiple logging points may increase code added to the software system to capture the event logs. The code may take some time to write and maintain. This can increase the structural and behavioural complexity of the code throughout its life cycle.
System resource impact	With the additional code needed to capture the logs, the usage of system resources, such as central processing unit (CPU) and I/O (input/output) channels, will increase. This may negatively impact the performance of existing system operations or increase the cost of keeping the system at the same operating speed by increasing the system resources.
Unusable logs	Adding numerous logging points or logging too many at points can produce numerous trivial or useless logs that will not improve system utilisation analysis. When implementing the system utilisation analysis stage, the logs might need to be filtered or modified to be more meaningful. As many as 70% of the logs may be irrelevant [44]. The software engineers and developers write the logs, which can sometimes be irrelevant to other managers or system administrators when implementing a log analysis report. More event logs can have missing or incomplete logging attributes due to the additional logging points. The increase in the behavioural complexity of the log can impact the decisions made to improve software maintenance.

  The accuracy and trustworthiness of the event log are more important than capturing the many available event logs in a software system [42, 45]. The additional unnecessary logs will also take up more storage space, increasing costs and possibly influencing the software system's performance.

- **Event log complexities**

  Software always involves complexity that increases as the software system becomes more considerable. For the event log quality model, the complexity of the event can be split into two different complexities, namely [43]:

  - Structural complexity is the application of different algorithms in the software that allows the event log to be evaluated when it occurs, which can alter the behaviour of the event log.
  - Behavioural complexity is the complexity of the behaviour of the event logs that refers to the number of smaller events in each captured trace and the different variations of these traces within an event log.

  These two complexity attributes can be costly when the event logging mechanism needs to be constantly maintained in large software systems, which can affect the rest of the system's performance or the integrity of the captured event logs [9]. The constant modification of the event log software can be due to technical debt, as the complexities of the event log system lead to technical issues when attempting to log an event or when they are not compatible with other systems [6].

- **Consistency of event logging**

  The accuracy and consistency of event logs are critical to making reliable decisions based on the identified behaviour of the software system and the historical data in the event logs, which are discussed in the previous dimension of event log quality [16, 43]. With the accuracy and trustworthiness of the event logs properly applied, their consistency should be accepted as correct and verifiable when comparing them to the software system.

  An event log quality model is essential to ensure that the logs are high quality for the log analysis data mining process; therefore, the event log data should be consistent. To ensure the consistency of the event logs, the structure of the event logging points and log parsers should be compatible with capturing all the critical log attributes. The event log data should also be consistently analysed with different methods used as part of the consistency of the event logging process.

- **Completeness of event logging**

  The event logs will be analysed later; the logs should be fully complete when used, as some of the other logging attributes might not be available at that stage. The event logs' available attributes should be accurately captured before storing them in a database to ensure that no missing event data or missing events are discarded due to incomplete information. Two types of completeness attributes exclude the availability

attribute in Figure 1.5:

- Local completeness refers to all event data that can be captured for an instance of the event that takes place that can be added as a log attribute to the event log [43, 46].
- Global completeness refers to the occurrence of all possible outcomes or behaviours of the event logs that can be captured, which is required for the system utilisation analysis [43, 46].

Ensuring that both completeness levels are achieved and that the event log data is complete can impact performance if specific data are not directly available during the instance when the event has occurred. The logging mechanism must capture this as efficiently as possible without causing performance issues to the rest of the software system's operations [32, 42].

Logging attributes

Before the logs can be parsed to a structured data set, the key attributes must be defined for the event log [47]. The attributes describing the event log in Table 1.5 are the essential attributes a log event should have.

Table 1.5: *Basic log event attributes [47]*

Attribute	Description
Case number	Unique identifiers for each log event. This is usually the primary identifier of the log event.
Timestamp	The time and date that the log event occurred. This is part of identifying the order of events or traces along with the case number of the event log [43].
Event type	Each log event can be grouped with other log events with similar actions. These event-type attributes should be classified based on a state change, failure to execute an instruction, or an activity occurrence, such as service availability [44]. The event type is usually also the log level. The log level in event logging reflects the severity of the event log [48]. An event of interest may have different log levels, making it easier to capture the kinds of events software engineers and developers are interested in.
Originator	The origin of the event in the software system. This can be parts of the software that perform the event action or was the cause for the event to be initiated by another part of the software.
Other metadata	This is any other relevant information that can be used as the event log's attribute that further expands the information of the log event. This can be an additional field or many other individual attribute fields.

Logging parsing and log points

Knowing what to log can significantly reduce any overhead the logging mechanism may produce in the software system [34, 49]. Preserving quality (as described in Table 1.3) is necessary to ensure that the obtained logs fulfil their purpose when analysed.

These attributes make it possible to mine and analyse the logs and increase the precision and reliability of the event log [43]. The case number and timestamp attributes in Table 1.5 can be defined anytime during the logging process. This is not the case for the rest of the attributes.

Every log should have a defined action that will sort it into a group of logs that can be defined as the type of event. These event types can be predefined by what is expected from the event action or will need to be observed later in the analysis of the logs in case there is no explicit grouping of the logs [44, 47].

Log points

The sources of the log event assist in determining where the event took place. For event logs, it is essential to recreate scenarios or actions based on the relevant parts of the software system that participated in the event action.

Other metadata can increase the quality of the log by providing additional information about the executed software instructions. These attributes add more information that can be used to recreate the scenario or action that may be unique parameters or other events that participated in the event log.

Obtaining the attributes in Table 1.5, an instruction generates the log and parses it onto a data set. These log instructions are called logging points in the software environment [34,42]. They can be any instruction, such as a print function that displays the information for the user to more complex processes or libraries that third-party developers can create. Figure 1.6 is an example of a logging point parsing a log message in a structured log.

```
/* A logging code snippet extracted from:
    hadoop/hfs/server/datanode/BlockReceiver.java */

LOG.info("Received block " + block + " of size "
     + block.getNumBytes() + " from " + inAddr);
```

Log Message

```
2015-10-18 18:05:29,570 INFO dfs.DataNode$PacketResponder: Received
 block blk_-562725280853087685 of size 67108864 from /10.251.91.84
```

Structured Log

TIMESTAMP	2015-10-18 18:05:29,570
LEVEL	INFO
COMPONENT	dfs.DataNode$PacketResponder
EVENT TEMPLATE	Received block <*> of size <*> from /<*>
PARAMETERS	["blk_-562725280853087685", "67108864", "10.251.91.84"]

Figure 1.6: *An illustrative example of log parsing [32]*

The defined attributes are captured by the logging point when the event takes place or occurs. The created log message is then parsed into a structured log to be safe in a database or displayed. The logging point should be strategically placed to capture the attributes required to complete the log event [44].

Determine where to place the logging point in software directly affected by the attributes and whether the captured log will be high quality as described in Table 1.3. The availability of consistent, high-quality logs will directly impact the process of mining the logs when analysing them [43].

To strategically place a logging point, developers should consider what the activation of the logging point will be during the run time of the software system [29, 34]. The activation can be straightforward if a statement meets specific criteria or instructions that execute after an event or action occurs (e.g. run-time errors).

The log level or event type outlined in Table 1.5 can be used to determine where the log points should be placed. The aim of establishing the logging point should be to capture all the log attributes when the event of interest has occurred.

Log analysis in web-based applications

With the log parsing and log points defined in Table 1.3, a log analysis can be made from the stored event logs. The log analysis is the data mining process focused on the software system's generated event logs [50, 51].

The defined log attributes outlined in Table 1.5 will be used to complete the log analysis. Each software system will have some variation of the log attributes in Table 1.5. For a web-based application, the log attributes will also contain data about the requests between a Web server and a Web client and its responses [50, 52]. This will also reflect on the log level at which weblogs are obtained for log analysis.

1.2.3 Log analysis

Gathering usage statistics and user behaviour data in web-based applications is called Web analytics [53]. Web analytics can be used for user modelling efforts and is a form of log analysis. User modelling in software engineering is the customisation and adaptation of software systems to the users' required needs [54, 55].

User modelling can also include the implementation of software maintenance, as maintenance is an adaptation of the software system to the user's needs, which is the utilisation analysis using event logs. The web analytics for this study focuses on different analytics, as seen in Table 1.6.

Table 1.6: *Web analytic for user-based data*

Analytic	Description
Identity of the user	This is any information about the user's identity in the software system. Users can have different roles when using the software system, such as a system admin or general user. These roles mostly dictate what the user can access and do on a website.
Site interaction	The different Websites the user is accessing during their active Web session. This would also contain all the information about the following: • how often the users visit a website, • how much time they spent on a specific website, and • navigation between different web pages of the website.

These same analytics can be used for none-web-based applications because the:

- identity of the user can be captured if the software system uses a software license and
- different parts of the system can be tracked, as well as services the user uses.

Analytic tools for event logs

Numerous third-party analytic tools for event log data monitoring and management to visualise log data graphically. Choosing the correct device to use can depend on what the software engineers and developers want to analyse and the availability of the tools due to external factors such as cost and usability.

Event logging monitoring and management tools are sometimes underused and are often not used to their full potential when analysing event logs [44]. This can be due to the log quality of event logs not meeting the standards of Figure 1.5.

1.2.4 Artifical intelligence and automated logging

Automated logging through AI (Artificial Intelligence) offers the creation of logging points for software [56]. The primary goal is to simplify the logging process, reduce dependence on developers' expertise, and tailor logs for automated methods. While this approach brings various advantages, it also presents challenges such as concerns about performance overhead, system integrity, and quality assessment [32, 56].

In this study, AI-based logging approaches will not be considered over traditional logging mechanisms due to the following reasons:

- **Simplicity:** Traditional logging mechanisms provide a straightforward and well-defined approach to capturing user-based activity logs without the complexity of AI algorithms and models.
- **Control:** Developers retain direct control over the attributes and data captured in logs with a traditional logging mechanism, enabling specific customisation based on the application's requirements.
- **Resource Efficiency:** Implementing a traditional logging mechanism may require fewer computational resources than AI-based approaches, making it more suitable for specific applications or environments. Costs are also a consideration, as more specialised AI-based tools can be expensive.
- **Compatibility:** Traditional logging mechanisms enjoy widespread support and easy integration into existing software systems without needing specialised AI infrastructure.

1.2.5 Gap identification

Section 1.1 emphasises that software maintenance is an important stage in the SDLC of software systems. Implementing software maintenance efficiently with the limited available resources organisations or individuals have is crucial to improving the efficiency of software maintenance efforts. Assisting developers in making better-informed decisions about which systems to prioritise maintenance can improve software maintenance.

State of the art topics

From the literature, there were a few critical focus points identified to create a log-in mechanism for user-based activities. These focus points exist for the research done in Sections 1.2.1 to 1.2.3.

The research for this study focused on accredited peer-reviewed published journals or articles for the last two decades (2000-2022). Some exceptions include literature older than 20 years since software maintenance has been essential to modern software development over the past few decades. These studies were obtained from journals in the IEEE Digital Library by focussing on three key topics: software maintenance, event logging, and log analysis. Each of the three primary states of the art topics is divided into subtopics that explore different studies relevant to the main issue.

Software maintenance

Software maintenance implementation in industry and best practises when implementing subtopics of software maintenance are described in Table 1.7.

The three different state of the art subtopics for software maintenance in Table 1.7 evaluate the studies that discuss software maintenance and software maintenance implementation. These subtopics focus on what is needed to implement software maintenance for any given software environment using industry standards.

Table 1.7: *Software maintenance state of the art sub-topics*

Topic	Description	Evaluation criteria
Models	Software maintenance models and implementation in industry.	1. Did the study focus on software maintenance? 2. Did the study discuss different software maintenance implementations?
Problems	Challenges with implementing software maintenance.	1. Did the study identify frequent problems with implementing software maintenance?
Prioritisation	Prioritisation methods.	1. Did the study discuss the importance of prioritisation techniques of certain software maintenance activities, models, or software components?

Event logging

Event logging in a software environment and activities implementing state of the art event logging subtopics are described in Table 1.8.

The event logging subtopics aim to create or use a suitable logging mechanism for the log analysis requirements through the captured logging points. Table 1.8 identifies studies that will adhere to the best industry practises to create the logging mechanism and use or create the design methodology needed for the logging mechanism.

Table 1.8: *Event logging state of the art topics*

Topic	Description	Evaluation criteria
Points	Software logging points and log attribute identification	1. Did the study discuss what needs to be logged or how to identify key logging attributes from certain software events? 2. Did the study aim to create or identify key logging points for specific log analysis purposes?
Parsing	Challenges with implementing software maintenance.	1. Did the study aim to create or identify key logging points for specific log analysis purposes in a software environment? 2. Did the study provide a method or standards for log parsing to adhere to a log quality model?

Log analysis

A log analysis of software systems using state of the art event logging topics is described in Table 1.9.

Table 1.9: *A log analysis using state of the art topics*

Topic	Description	Evaluation criteria
Utilisation	Utilisation log analysis for user-based event logging.	1. Is a utilisation analysis done? 2. Is prioritisation done based on the utilisation log analogy?

In Table 1.9, the topics focus more on studies that implement a log analysis for utilisation purposes. Tracking certain events is therefore essential for log analysis to ensure that the precision criteria of log quality can be achieved by Table 1.3 to efficiently use or create a log analysis report for software maintenance.

State of the art summary

Using the defined criteria created for the state of the art topics in Tables 1.7 to 1.9, state of the art research for the obtained studies is done in Table 1.10.

Table 1.10: *State of the art*

Ref.	Software maintenance			Event logging		Log analysis
	Models	Problems	Prioritisation	Parsing	Points	Utilisation
[9]	Partial	✓	✓	✘	✘	✘
[10]	Partial	Partial	✓	✘	✘	✘
[12]	Partial	✓	✓	✘	✘	✘
[16]	✘	✓	✓	✘	✘	✘
[18]	✓	Partial	✘	✘	✘	✘
[24]	Partial	✓	✓	✘	✘	✘
[28]	Partial	✘	✓	✘	✘	✘
[32]	✘	✘	✘	✓	Partial	✘
[56]	✘	✘	✘	✓	✓	✓
[35]	✘	✘	✘	Partial	Partial	✘
[42]	✘	✘	✘	✓	✓	✘
[43]	✘	✘	✘	Partial	Partial	✘
[51]	✘	✘	✘	✓	✓	Partial
[50]	✘	✘	✘	✓	✓	Partial
[54]	✘	✘	✘	✘	✘	✓
[57]	✘	✘	✘	✘	✘	✓

In Table 1.10, studies marked with partial status satisfied the subtopic evaluation criteria to some extent. These studies still contained much-needed literature for the subtopic, even if it was marked as practically meeting the evaluation criteria.

In Table 1.10, the three main topics split the obtained literature into two main parts. The first part comprises software maintenance and how to implement and prioritise it using a suitable software maintenance model. The second part includes event logging and log analysis.

Review of the state of art studies

This section describes the state of the art studies identified in Table 1.10 and provides commentary on the relevance of each to this study.

On the Relationship between Software Complexity and Maintenance Costs [9]	
Study summary:	Defines the different problems with the implementation of software maintenance and its complexities as described in Section 1 (Maintenance Problems).
Method:	Three different software operating systems were utilised to compare estimated development costs, maintenance costs, code lines, and code complexity. This is used in the analysis to determine the relationship between the complexity of the software and the maintenance costs.
Results:	The results showed that the estimated maintenance costs are such that the complexity of the software needs management to reduce these costs.
Commentary:	The study is used in this review of the literature when discussing the importance of software maintenance difficulties in the implementation of a software maintenance model. The study does not provide solutions on how to implement software maintenance effectively. It explores the strategies and best practices in the literature review to help with software maintenance resource costs.
Study keywords: Software; Software maintenance; Software evolution; Maintenance costs; Software evolution; Software maintenance	

Metric-based tracking management in software maintenance [10]	
Study summary:	Explores the implementation of software maintenance with role-players involved.
Method:	Assigned roles of each individual in the software maintenance process (user, coordinator, decision maker, and maintenance operator) and evaluated their impact on the software maintenance process.
Results:	Results indicate explicit management issues when implementing software maintenance.
Commentary:	The study recommends implementing a practical software maintenance model to solve management problems and make maintenance processes more efficient.
Study keywords:	Metric; Software maintenance; Tracking management

A cost model for software maintenance & evolution [12]	
Study summary:	Examines the cost of software maintenance implementation and investigates the difficulties with software maintenance and the evolution of maintenance models.
Method:	Attempts to assess the cost involved for a software system by implementing a prediction cost analysis of multiple types of maintenance activities.
Results:	Shows that estimated maintenance costs depend on factors like the size of the development team, code complexity, and code quality.
Commentary:	Highlights the need for efficient software maintenance but doesn't provide solutions. Offers insights into possible resource costs when implementing a software maintenance model.
Study keywords:	Maintenance cost estimation; Software life cycle costing models; Software Maintenance and evolution; Software product management

Trends in software maintenance tasks distribution among programmers: A study in a micro software company [16]

Study summary:	Examines software maintenance efforts when implementing a software maintenance model. Investigate how software developers prioritise and distribute maintenance tasks.
Method:	Classifies maintenance tasks into different types and distributes them among developers using multiple test distribution techniques.
Results:	Effective distribution techniques divide maintenance tasks between developers and improve study scheduling.
Commentary:	It focuses on the contribution of maintenance tasks by developers and how efficiently they distribute the work among themselves. It primarily looks at decision-making based on maintenance tasks obtained by developers.

Study keywords: Maintenance Engineering, Market research, Companies, Software maintenance

Supporting Software Architecture Maintenance by Providing Task-specific Recommendations [18]

Study summary:	Discusses various types of maintenance in the industry and the need for software maintenance. Defines maintenance types and associated activities for operations and maintenance.
Method:	It aims to address developers' lack of information when implementing maintenance.
Results:	Lists multiple solutions for software maintenance recommendations.
Commentary:	Examines the software maintenance process in the industry, focusing on maintenance tasks and the correct approach to efficient implementation rather than improvement.

Study keywords: Software maintenance; Natural language processing; Software architecture; Text classification

Analysing Forty Years of Software Maintenance Models [24]	
Study summary:	Analyses software maintenance models and their characteristics over the last four decades.
Method:	Obtained studies and systematically analysed 1,044 articles on software maintenance models from 1970 to 2015.
Results:	Identifies common aspects and problems in the software maintenance industry, including limited third-party validation, lack of improvement on existing models, limited literature comparison, and challenges in replicability due to private data sets and custom tools.
Commentary:	Focuses on highlighting the limitations of software maintenance in the industry rather than providing strategies to improve it. Discusses issues such as slow adaptability of maintenance models, lack of literature comparison due to closed-source models, and challenges in replicating research studies.
Study keywords:	Software maintenance; Systematic review of the literature.

A Software Maintenance Methodology: An Approach Applied to Software Aging [28]	
Study summary:	Discusses the implementation of suitable software maintenance models for software maintenance activities by characterising the software system.
Method:	Implements a software rejuvenation method by performing maintenance on specific systems and evaluates the performance improvement of these older systems.
Results:	Shows that applying the software rejuvenation method to specific systems improves system performance.
Commentary:	Focuses on software maintenance strategies for older systems without providing a guideline for efficient maintenance implementation. Addresses the use of maintenance methods for such systems.
Study keywords:	Software ageing and rejuvenation; Methodology; Software maintenance

Tools and Benchmarks for Automated Log Parsing [32]	
Study summary:	Focuses on log parsing and benchmarking different log parsing methods.
Method:	Compares multiple third-party log parsers using various benchmarks.
Results:	Demonstrates the robustness, efficiency, and accuracy of these log parsers through benchmarking data.
Commentary:	Emphasizes the importance of efficiency in event logging. Primarily addresses log parsers for system diagnostics rather than user-based event logging.
Study keywords:	AIOps; anomaly detection; Log analysis; Log management; Log parsing

Auto-Logging: AI-centred Logging Instrumentation [56]	
Study summary:	This study introduces auto-logging, which automates the process of logging to better suit AI-enabled methods as end-log consumers.
Method:	The methodology involves formulating the dynamic strategy as a sequential decision-making problem and applying ideas from reinforcement machine learning to learn the automated logging method.
Results:	The study provides the concept of automated logging and discusses the instrumentation related to it.
Commentary:	The idea of automated logging challenges the traditional logging mechanisms by providing advantages of it towards it. The potential benefits of adapting logging for AI-enabled methods is discussed but the amount if needed resources and complexity of using it is not needed for a simplicitic logging mechanism.
Study keywords:	Software Engineering, Logging, AIOps

A Systematic Review of Logging Practice in Software Engineering [35]

Study summary:	Examines how logging practices are implemented in the industry and highlights the lack of research in software engineering.
Method:	Created multiple research equations to obtain and analyse research on logging practices in software engineering.
Results:	Shows that relevant research has been conducted each year, categorising the development topics of each article.
Commentary:	Emphasizes the need to establish comprehensive methods for developers to implement logging practices. Highlights the lack of sufficient research on logging practices and the importance of specific use cases in the software industry.

Study keywords: Logging practice; Software engineering; Systematic literature review

Learning to Log: Helping Developers Make Informed Logging Decisions [42]

Study summary:	Summarises the importance of how and what to log to assist software developers in creating efficient logging mechanisms.
Method:	We have developed an automated tool called "LogAdvisor" for developers, trained using different learning models.
Results:	Presents benchmark results of the tool's performance with various learning models.
Commentary:	Provides a generic method for log analysis of events based on intended log analysis. Offers important guidelines for event logging and log analysis, which should be incorporated into the process of a new logging mechanism.

Study keywords: Keywords (not specified in the provided text)

Towards a Better Assessment of Event Logs Quality [43]	
Study summary:	Examines development practices that can be implemented to improve or maintain the quality of event logs.
Method:	Created metrics for the quality assessment of event logs in real-life and artificial case studies.
Results:	Validated the quality metrics of event logs using natural and synthetic data.
Commentary:	Highlights the importance of maintaining acceptable log quality to ensure consistency and completeness of logs. Emphasizes the need for logging mechanisms to capture logs with minimal structural and behavioural complexity accurately.
Study keywords:	Event logs; Process mining; Process mining algorithms; Qualitative model

Central Audit Logging Mechanism in Personal Data Web Services [51]	
Study summary:	Aims to create a logging mechanism to obtain audit logs from a web application.
Method:	Created a central logging mechanism for audit logs in web services.
Results:	Demonstrated using an applied logging model to establish a central audit logging mechanism for a web service.
Commentary:	Created a method for audit logs in a web service but did not perform log analysis on the logs. Explored the advantages and disadvantages of the created logging mechanism and discussed the differences between client- and server-side logging mechanisms.
Study keywords:	API; API policy; Audit logging; Personal data; Web service

User Behavioural Patterns and Reduced User Profiles Extracted from Log Files [50]	
Study summary:	It focuses on extracting behavioural patterns from log files through log analysis and creating user profiles for comparison with individual users or user groups.
Method:	Analyses behavioural patterns in log files generated by users interacting with a system. Aims to identify similarities between logs based on specific log attributes.
Results:	Demonstrates similarities between users or user groups when they interact with the system.
Commentary:	This study does not provide a specific logging mechanism method for implementation on any system. Instead, it conducts log analysis to create user profiles and explores what events in a system can be considered user-based events for log analysis.
Study keywords:	Analysis of users' behaviour; behavioural patterns; Complex networks; User profiles

Tracking User Activities and Marketplace Dynamics in Classified Advertisements [54]	
Study summary:	Focuses on tracking user activities on a web application when users interact with advertisements. Implements log analysis to create a probabilistic model of user behaviour data based on advertisement interactions.
Method:	Creates models to track user activities when interacting with web page advertisements.
Results:	Demonstrates the performance and evaluation of each model used.
Commentary:	This study explicitly records user interactions with advertisements for log analysis, highlighting the goal of capturing user-based data for marketing profiling. This reflects a common objective in the industry where organisations track user data for commercial purposes.
Study keywords:	Classified ads; Temporal analysis; User modelling; User tracking

Analysis of Visitor's Behavior from Web Log using WebLog Expert Tool [57]	
Study summary:	Explains website log analysis using a weblog tool that performs data processing, pattern discovery, and research. Aims to obtain user activity data, including total time spent per webpage, visited webpages, and other visitor browser analysis data.
Method:	Utilises a weblog tool to capture specific event logs for web log analysis of user behaviour.
Results:	Indicates that the log data used in the analysis could be leveraged to enhance the website's usability.
Commentary:	It focuses on log analysis of user data rather than providing a method for the logging mechanism itself.
Study keywords:	web server log; Web usage mining

1.3 Problem statement

Software maintenance is a vital part of the entire life cycle of any software system. Section 1.2.1 discussed why limited available resources make it more challenging to improve maintenance efforts. It is beneficial to prioritise maintenance to ensure that limited resources are used efficiently.

A possible solution to this problem is using event logs to determine system usage due to event logging being a proven method in the industry to track system usage. Software developers can access third-party software logging tools to get the event logs. The majority of tools focus on system runtime utilisation rather than user activities. Logging tools that track user-based movements do exist, depending on the framework the software system is developed.

Although there are live tools that track user-generated events, it is not guaranteed that log quality will be acceptable. Important logging attributes that will aid log analysis may not be logged.

Designing and implementing a logging mechanism could bridge the gap between software maintenance and log analysis to create a system usage report.

Software developers must still design the overall logging mechanism and decide where to place the logging points to capture the event logs in a software system. There are proven methods to create a suitable logging mechanism, but many do not include analysing records for user-based utilisation to improve software maintenance. The problem statement can be summarised as follows:

> *Software maintenance is a problem in the industry due to how inefficiently developers prioritise maintenance activities. A proven method to monitor software behaviours is event logging. The logging mechanism and log analysis must be explicitly designed for user-based events to improve software maintenance.*

1.4 Objectives of the study

In Section 1.3, the problem of efficiently implementing software maintenance has been identified. There is a need to create a system utilisation analysis report from event logs captured using a logging mechanism.

The study is divided into two components to achieve the primary objective: design and implement the logarithmic mechanism.

1.4.1 Literature Objectives:

1. Investigate what will define a user-based event log and how to obtain the needed log attributes for a comprehensive user utilisation log analysis.
2. Investigate where to place logging points in a software system to capture user-based event logs.
3. Investigate how to evaluate the log quality of the stored user-based event logs.
4. Investigate how to implement maintenance prioritisation.

1.4.2 Empirical objectives:

1. Implement the method to create a user-based logging mechanism on a test system:
   (a) Define the user activity types and create the needed log attributes for a user-based event log.
   (b) Implement the logging points at strategic locations to capture the relevant log attributes to create event logs.
   (c) Implement a software maintenance prioritisation method in the log analysis.
   (d) Use or create a log analysis tool to evaluate the log quality of the captured logs.
2. Verify the results for the test system's software maintenance prioritisation.
3. Validate the software maintenance prioritisation method by critically analysing the case studies' results.

1.5 Overview of the dissertation

Chapter 1: Introduction

This chapter discusses the background of software maintenance, event logging, and system utilisation analysis. It defines the complexities and general issues of software maintenance for software developers. Efficient use of limited resources requires software developers to implement better software maintenance decisions. Event logging is a proven method to gather system information that can assist with prioritising software maintenance to implement it efficiently.

Chapter 2: Methodology

This chapter discusses the design of the generic method used to create a logging mechanism from a set of defined logging points and attributes. The software system for creating the logging mechanism is a web-based application. The second part of this chapter is the system utilisation analysis, using the captured logs to generate an analysis report.

Chapter 3: Results

This chapter contains the results of implementing a logging mechanism in a case study web-based application, as designed in Chapter 2. The results of the implemented logging mechanism are analysed as part of a system utilisation analysis, and recommendations are made on improving the maintenance of the case study web-based application. The results are discussed and validated to show how they address the problem statement and fulfil the study objectives.

Chapter 4: Conclusion

This chapter concludes by creating a logging mechanism for system utilisation analysis to improve software maintenance for a web-based application. Limitations and recommendations are also identified based on the methodology and results.

Chapter 2

Methodology

2.1 Preamble

The literature in Chapter 1 informed the method to create a logging mechanism that captures user-based activity logs to improve software maintenance by analysing the logs obtained. The development of the solution to the identified problem in Section 1.3 will be explicitly made for a Web-based application.

Web-based applications are one of the most widely used software implementations that benefit from analysing system usage using user-based event logs. These software systems have many different designs. For this study, web applications that require the user to be logged in to an active session in the software system will be used, as seen in Figure 2.1.

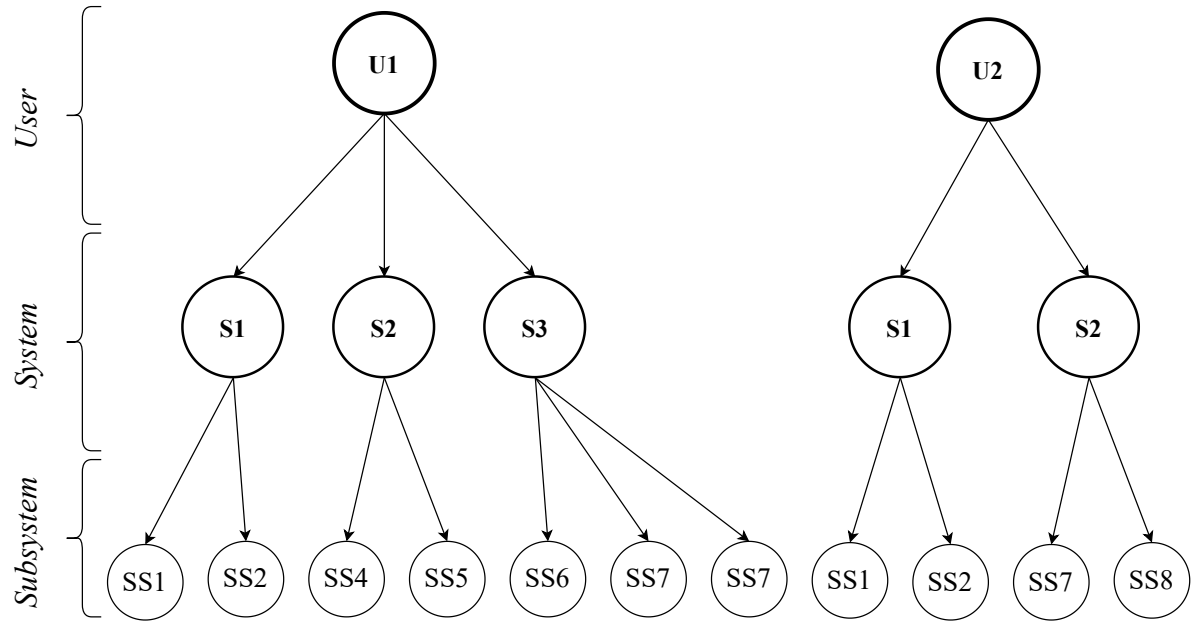

Figure 2.1: *Basic design of a software system*

Figure 2.1 is the basic design of the web application users interact with in different systems. Examples of these systems include the various pages or sections of the website and can be further divided into subsystems. Section 2.3 discusses the method to create a logging mechanism to capture user-generated events is discussed for web-based applications. The different functional requirements and interfaces are also discussed in this section [58].

Sections 2.3.3 and 2.3.4 discuss the method used to analyse the obtained logs to improve software maintenance using various visualisation tools or available log attributes. Figure 2.2 illustrates the design for the logging mechanism to capture the user-based event logs.

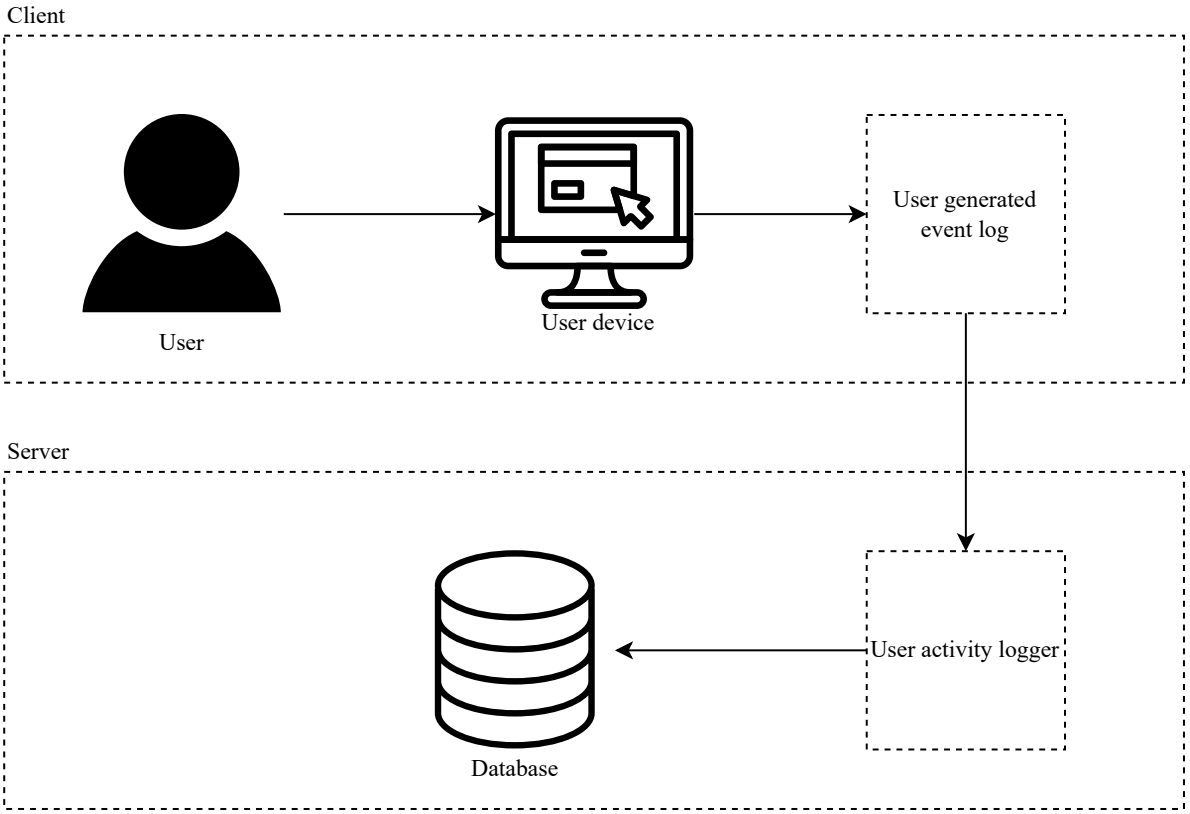

Figure 2.2: *Logging mechanism basic system design*

Figure 2.2 illustrates the basic system design of the logging mechanism. This design provides a high-level overview of the user's interaction with the software system to create a user-based event log. The two sides of the design in Figure 2.2 compromise:

- The client side involves the user, the user's device to access the website and the user-generated event. The user interacts with the website, creating a user event that can be captured.
- The server side has the rest of the user activity logging software consisting of single or multiple logging points. The user activity logger interacts with a structured database to store the captured log created from the captured log attributes.

2.2 Development of the solution

For this study, the development of the solution comprises four main phases that were instrumental in guiding the research design process [59]:

- *Investigate* use methods in the literature or create new methods for a logging mechanism and log-maintenance prioritisation analysis,
- *design* a logging mechanism that performs software prioritisation log analysis,
- *verify* whether the solution meets the design specifications and
- *implement* the solution in case studies to evaluate the results.

Figure 2.3 illustrates how these four development parts of the solution are used to create the methodology for this study.

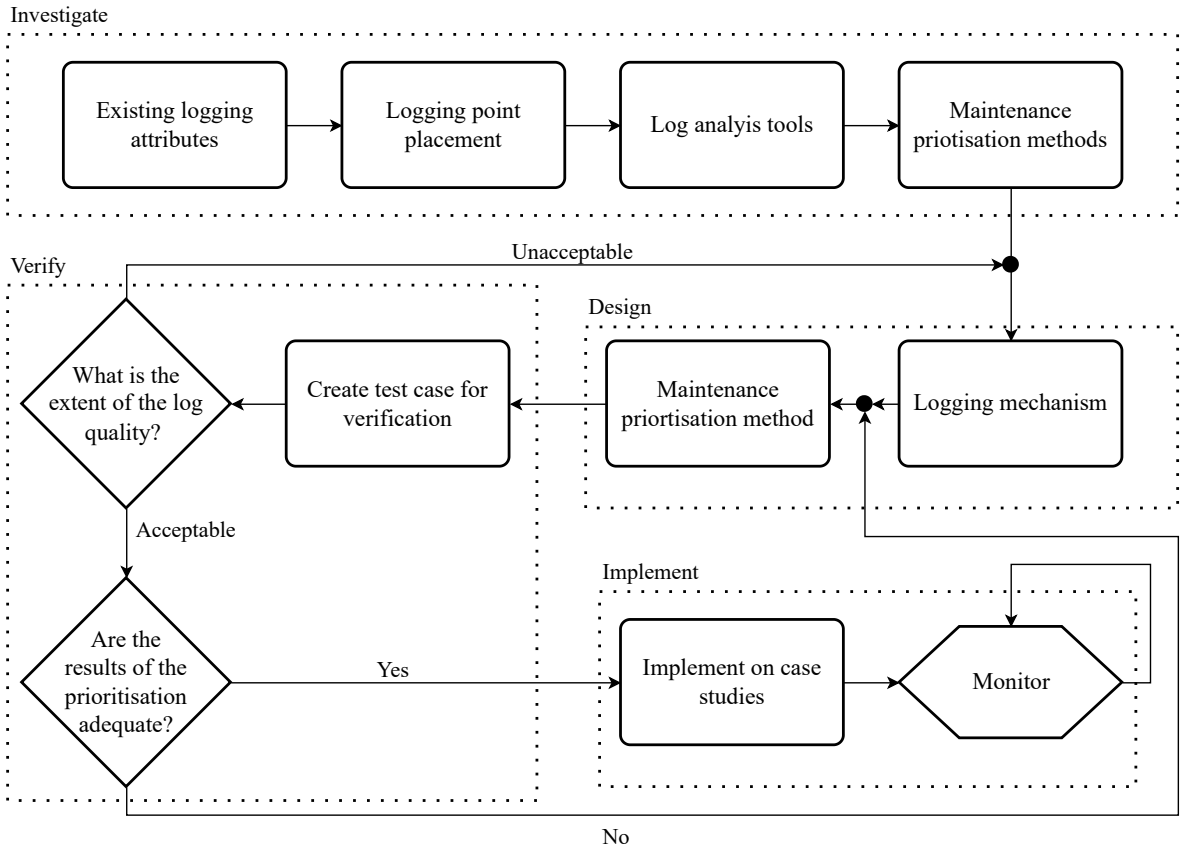

Figure 2.3: *Development of solution*

The first phase in Figure 2.3 is the investigation in Section 2.3 to define the requirements for the logging mechanism and the prioritisation of software maintenance using the literature outlined in Section 1.2. Section 2.4 discusses the second phase, the designing of the logging mechanism, and the prioritising of the software maintenance.

The verification phase focuses on implementing the design specifications of the logarithmic machinery method and the software maintenance method. The implementation phase follows the verification phase, whereby the methods are implemented on the case studies to obtain further monitored and discussed results.

2.3 Investigate

In this section, functional requirements will be created for the logging mechanism and prioritisation of software maintenance. Table 2.1 summarises each of the main functional development requirements created from the investigation phase in Figure 2.3.

The functional requirements (**F/R**) in Table 2.1 are used to identify each requirement for the development of a solution that can later be verified. Sub-requirements will use the same labelling.

Table 2.1: *Functional requirements of the solution*

Req. ID	Name	Description
F/R 1	Identify log attributes	The logarithmic attributes must be identified to complete the log analysis to prioritise software maintenance. The log attributes form the characteristics of the obtained event, herein referred to as user-based events.
F/R 2	Logging point creation	Logging points capture the desired log attributes when a user-based event occurs. In Table 1.3, logging points should be strategically placed in the software system to capture the event logs. This stage of development includes identifying where log points should be placed, creating logs, and interacting with the database.
F/R 3	Log analysis tools	Use suitable third-party tools or create software to analyse the stored event logs.

Continued on the next page

Table 2.1: *(continued from the previous page)*

Req. ID	Name	Description
F/R 4	Maintenance prioritising	Create a maintenance prioritising report based on the log analysis for software maintenance using the log analysis tool. The report aims to visualise the use of certain system parts and determine maintenance prioritisation.

The design of the logging mechanism system will be defined in this section to create a generic method for creating a suitable logging mechanism to capture specific user-based event logs. Additionally, this defines any other subfunctional requirements to identify log attributes (F/R 1) and the creation of log points (F/R 2).

2.3.1 Log attribute requirements

The functional requirement of log attributes (F/R 1) focuses on the possible events involving the user when interacting with the software system. Table 2.2 outlines the functional requirements of F/R 1.

Table 2.2: *Log attributes functional requirements (F/R 1)*

Req. ID	Name	Description
F/R 1.1	User-based event log characteristics	This functional requirement defines the characteristics of what software system events can be classified as a user-based event.
F/R 1.2	User activity types	User activity types further expand on what event types are valid for F/R 1.1. This is also the first categorisation of the logs obtained.
F/R 1.3	Log attributes	The log attributes are the data obtained that describe a log of user-based events. These are the primary data that will be stored in a structured database.

Table 2.2 is the first functional requirement for developing a solution that must be defined. It is essential to define and create the characteristics of an event log for a user-based event log.

User-based event log characteristics

The user is the initiator of the logging mechanism. Each action or event they trigger by interacting with the user interface on their device can be a potential user-generated event. Table 2.3 outlines the sub-requirements for the user that the event log should fulfil to be classified as a user-based activity log.

Table 2.3: *Requirements for an event to be classified as a user-based activity*

Req. ID	Description
F/R 1.1.1	The event must be triggered by the user interacting with the user interface using their device and not any other events that the system self-initiates. The user needs to interact directly with the UI. This can also be validated by tracking whether the user did interact with the UI of the HyperText Markup Language (HTML) element IDs.
F/R 1.1.2	The event must consist of different cases ($ca \in CA$ the cases consist of events) that are noteworthy to make the event log identifiable [50].
F/R 1.1.3	For certain types of event logs for F/R 1.1.2, the user-generated event should have an origin from which the event took place.
F/R 1.1.4	The event log should consist of attributes that expand the identity of the user-based activity.
F/R 1.1.5	The event must have the user as the initiator or input for the user-based activity. This will exclude all events the system triggers, as the user did not initiate the event directly.
F/R 1.1.6	Only use the first *HTTP request*[1] that is sent to the server.

Every interaction the user has with the user interface of the device to the software system can be seen as an event triggered by the user. Most of these events will not have a meaningful impact as they will not fulfil F/R 1.1.2 and F/R 1.1.4 in Table 2.3.

[1] A **HTTP request** is made by a client to a named host, which is located on a server. The request aims to access a resource on the server.

For the user activity event to meet the requirement of F/R 1.1.2, it has to have defined cases describing each event's activity type. These activity types form the essential criteria for which events can be parsed, significantly reducing the number of logs that will be obtained. This will ensure that the event logging process will produce quality user-based logs as discussed in Table 1.3:

- A basic structural complexity to simplify log parsing and development of the logging points in the system,
- keep the logging consistent by not deviating from the defined cases and
- ensures that the event log's other attributes are complete and available to increase the accuracy and trustworthiness of the event logging when further system utilisation analysis needs to be done.

User activity types

The user activity types (F/R 1.2) categorise different user-based event logs when they are obtained before other log attributes are fully defined. Table 2.4 outlines the functional requirements of the basic user activity types.

Table 2.4: *User activity types*

Req. ID	Activity Type	Description
F/R 1.2.1	Web page accessed	The user may navigate through different web pages in a session. This tracks when the user first accessed a certain web page or software system on the page.
F/R 1.2.2	Session changes	This is any user activities excluding F/R 1.2.1 that modifies the user's session: • Logging into a web application, successful and failed attempted log-ins. This user-based activity may cause the log attributes that identify the user to be a `NULL` value as the user's session has not started yet, • Ending their session by logging out or declining to extend their session when it is about to expire, • Modifying any session or other relevant variables that can be used in the utilisation analysis
F/R 1.2.3	General activity	Any events excluding the first two types of user-based activity users initiate when interacting with the web page. Most user activity logs will have this event type.

The general type of user activity event (F/R 1.2.3) will be the most common user activity event and will be categorised into different user activity events.

These user activity types can be further expanded with the general activity (F/R 1.2.3) for log analysis purposes. The general activity types will differ for each system based on what the system enables the user to do or what is needed for further system utilisation analysis, such as determining whether the user's action was to generate a report they downloaded.

Log attributes

The log attributes (F/R 1.3) are the descriptive characteristics of the user-based event logs. The functional requirements for the log attributes are defined in Table 2.5.

Table 2.5: *Logging attributes*

Req. ID	Logging point	Description
F/R 1.3.1	Identification number	The activity identification is an incremental number of the user-based event that is logged.
F/R 1.3.2	Timestamp	This is when the user initiated the user-based activity event. This will be the timestamp from which the log was written in the database since the log will be made before the rest of the intended *HTTP request* is completed.
F/R 1.3.3	Activity type	Each event can be classified into user-based types. This is the type of activity based on users in Table 2.4.
F/R 1.3.4	User identification	Each user has a unique identification number that links the event to them if their session has been verified and can be obtained. It will not be available when the user tries to log in to the system as their session has not yet been set.
F/R 1.3.5	Request origin	In web applications, requests are always sent back to the server, which will call the primary function to handle the request. This can be logged as either the file from which the request is sent or the web page from which the request came.

Continued on the next page

Table 2.5: *(continued from the previous page)*

Req. ID	Logging point	Description
F/R 1.3.6	Metadata	The event metadata contain request parameters or other relevant request data. These metadata add more information about the user's activity. Some of the event types may not have metadata added.
F/R 1.3.7	Miscellaneous	These are any non-metadata attributes that can be consistently captured for use in the use analysis. They expand the log characteristics obtained from the user beyond the base attributes.

The logging attributes defined in Table 2.5 are the base attributes that make up the main structure of the user-based event log. For web-based applications on the client side, only some of these attributes can be obtained, as the rest of the attributes can be resolved on the server side. The metadata (F/R 1.3.6) may consist of the request parameters available on the server side, but any additional captured data can be added and sent to the server.

Each of these log attributes combined creates the base log from which key logging points can be created in the software system to capture user-based activity logs in Table 2.5. The activity type (F/R 1.3.3) can be assigned during the user-based activity identification phase with a default value and resolved to a new activity type based on metadata or other parameters by:

- Altering any of the session variables that are relevant to the system utilisation analysis,
- accessing a certain part of the software system that needs all the user-based activities set to a certain type based on the nature of the procedures that need to be executed, such as triggering a generation of a report that can be its user-based activity type, and
- sorting the activity type by HTML element tags, such as a button or text box.

Additional parameters may be captured on the client side, such as by the logging point, or they may be captured on the server side when the rest of the log's attributes are being obtained. These extra parameters are shown in Listing 2.1.

```
{ "RequestTarget" : "/Area4/Controller4/TestFunction",
  "RequestElementID" : "Button4",
  "RequestParameters": {
    "Parameter1": 4,
    "Parameter2": "Hello World!",
    "Parameter3": true
    "Parameter4": 40.404
    "Parameter5": {
      "Parameter6": "Car",
      "Parameter7" 160000.00
    }
  }
}
```

Listing 2.1: *Metadata JSON*

The metadata in Listing 2.1 represents possible additional parameters that can be obtained for the user-based activity log. The metadata will need to be stored as a JSON string because it can be a complex object that does not have a set number of parameters. This complex object can have:

- The `RequestTarget` parameter can be a file path for the website code from which the page is created or a software system. It also contains the function called by the *HTTP request*.
- The `RequestElementID` is the HTML element ID with which the user interacted, which caused the user-based activity. This can be used as additional validation that the user caused the event. Some user-based activities can ensure these HTML element types by getting the HTML element tag.
- The `RequestParameters` are all the parameters in the *HTTP request* that can be serialised into a JSON string. This can be used to determine what the user tried to do using this input for the specific function that is used for F/R 1.1.6 in Table 2.3.

2.3.2 Logging point requirements

The functional requirement of the logging point (F/R 2) focuses on creating and strategically placing the logarithmic points. The log points that obtain the log attributes must also store the created event log in a structured database. The logging point's subfunctional requirements are defined in Table 2.6.

Table 2.6: *Server functional requirements (F/R 2)*

Req. ID	Name	Description
F/R 2.1	Logging point placement	The logging points are used to capture and create the user-based event log that will be stored in a database.
F/R 2.2	Storing the user-based activity logs	The event log is stored in a structured database until it is needed for further log analysis.

Table 2.2 concludes the design requirements for the logging mechanism. The stored logs will be further processed when the log analysis needs to be implemented.

Logging point placement

In addition to Table 1.3, the logging points should be strategically placed in the software system to capture the log attributes for the user-based activity log. To meet the requirements of Table 2.3 for a user-based activity, the log points should adhere to the functional requirements of the log points of Table 2.7.

Table 2.7: *Logging point requirements*

Req. ID	Description
F/R 2.1.1	The logging point should be placed where the user's interaction with the software system will send a *request* back to the server.
F/R 2.1.2	Each logging point should consistently capture the user-based activity as the activity is happening.
F/R 2.1.3	Logging points should be globally complete to capture the user-based activities in the given software system without too much modification between each point in the same software system.
F/R 2.1.4	The logging points should not interfere with the rest of the system's operations; this would slow down the system by causing too much overhead in each *request* that is being sent.

The logging points can be either a single code segment or multiple code segments in a software system that aim to capture user-based actions as they happen. Creating multiple logging points in a software environment will:

- Increase the complexity of the logarithmic mechanism. Each point can be different from the other as it will need certain operations to capture the log,
- the consistency of the logging might differ and increase as the logging points increase in a software system and
- The correctness of the logging will be impacted if the different changes in the logging point cannot consistently capture the user-based activity or extract all the needed attributes to complete the user-based log.

Creating a single logging point reduces the complexity and, in most cases, will improve the consistency and correctness of the user-based logs. In web applications, a globally defined logging point can be used in a modified *HTTP request* that will form the base template for all or most *HTTP request* used in the software system, as discussed in Section 2.4.1.

Using a single centralised logging point does not guarantee that the logging mechanism will perform more efficiently and accurately than using multiple logging mechanisms. Using a single logging point may have complexity issues when the logging point needs to capture each user-based activity consistently with different cases.

Storing the user-based activity logs

After the logging point has captured the log attributes and created the event log, the log can be saved in a structured database. The storing of the functional requirements of user-based logs (F/R 2.2) should be defined for structured database interactions.

The captured parameters of the log attributes may have sensitive user data that should not be logged. Functions can be excluded or assigned a new user activity type that will need to filter out specific parameters or not log any parameters at all, such as:

- Session handling functions containing passwords or other user information should not be available to anyone but the user. This could lead to unintentional information disclosure of any personal information in the system utilisation analysis if it is available for anyone who can access and use the user-based activity logs and
- complex parameters such as file upload streams of files the user tries to upload. This information cannot be broken down to a simple `JSON` structure seen in Listing 2.1. Other metadata, such as the file size, name, and type, can rather be logged. Detecting these complex parameters allows these user-based activities to be defined separately.

Table 2.8 outlines the data type for the parameters and the functional requirements it will need to fulfil Table 2.5.

Table 2.8: *Log attributes for database storing of the event logs*

Req. ID	Column Name	Log attribute requirement
F/R 2.2.1	ActivityID	F/R 1.3.1
F/R 2.2.2	Timestamp	F/R 1.3.2
F/R 2.2.3	ActivityType	F/R 1.3.3
F/R 2.2.4	UserID	F/R 1.3.4
F/R 2.2.5	Subsystem	F/R 1.3.5
F/R 2.2.6	GroupID	F/R 1.3.7
F/R 2.2.7	MetaData	F/R 1.3.6

In Table 2.8, the functional requirements of log attributes should match the functional requirements (F/R 2.2) of the logging attribute (F/R 1.3). Any other parameters can be added using the miscellaneous (F/R 1.3.7).

New tables or other structured storage entities can be added to save the captured data from the logging point.

2.3.3 Log analysis tool

The log analysis (F/R 3) uses third-party analytical tools or custom-created software log analysis implementations. Developers should use the implementation that fulfils their log-analysis requirements. Table 2.9 is the log analysis functional requirements (F/R 3).

Table 2.9: *Log analysis functional requirements (F/R 3)*

Req. ID	Name	Description
F/R 3.1	Log quality	Log quality ensures that the logs obtained are usable for log analysis. Any incomplete logs should be either fixed in the log analysis implementation, or the logging mechanism should be adjusted to capture the complete logs.
F/R 3.2	Log analysis tool requirement	Certain basic requirements are needed for the log analysis tool to implement a log analysis.

Log quality

The log quality identified in Table 1.3 is significant when implementing a log-analysis mechanism. Log quality (F/R 3.1) will affect both the performance of the logging mechanism and the accuracy and completeness of the log quality. Table 2.10 outlines the functional requirements for log quality (F/R 3.1).

Table 2.10: *Log quality functional requirements (F/R 3.1)*

Req. ID	Requirement name	Description
F/R 3.1.1	Log availability	The log attributes should be available when the logging mechanism actively captures user-based events. It should be *locally-* and *globally complete*.
F/R 3.1.2	Log completeness	User-based logs should be complete, and minimal corrections should be made after logging during the log extraction process (F/R 3.1.3) and presentation of visualisation.
F/R 3.1.3	Log extraction	User-based logs are extracted from the database and imported into a visualisation presentation for user-based activity logs.

All functional requirements of the user-based activity log in Section 2.3.1 are achieved with minimal subsequent processing of the raw logs. Changes to the system that impact which possible user-based events are considered for logging are inevitable.

Log extraction refers to the methods used to obtain the logs from the database with any other relevant data that can be used in the visualisation presentation (F/R 3.2.1). The raw logs use foreign references to other tables in the database to provide more details on the user-based event log, as seen in Figure 2.7.

Log analysis tool requirement

The log analysis tool (F/R 3.2) ensures that a custom or third-party implementation is performed and able to meet some basic log-analysis requirements defined in Table 2.11.

Table 2.11: *Log analysis tool functional requirements (F/R 3.2)*

Req. ID	Requirement name	Description
F/R 3.2.1	Log visual presentation	The visual presentation of the extracted logs should be shown to the user who will use the activity logs in a custom visual system or other third-party tools. This will affect how the logs will be extracted (F/R 3.2.1) from the database because third-party systems may use an API to get the logs from the database.
F/R 3.2.2	Log comparison	Using F/R 3.2, use different log attributes used as defined criteria. This allows different types of users, subsystems, and activity types to be grouped and compared.
F/R 3.2.3	Maintenance suggestions	Maintenance suggestions can be made from system utilisation reports by prioritising maintenance or decommissioning software systems. This can be data or visual representations of the logarithmic comparison (F/R 3.2.2) using the log visual presentation systems (F/R 3.2.3) or creating a summary report from the visual presentation that contains maintenance suggestions.

Each functional requirement ensures that the system utilisation analysis for the created logging mechanism in Section 2.3 will be achieved. The primary user interface of the system usage analysis will consist of the presentation of user-based activities (F/R 3.2.1). This system will be either a custom-created system to display these logs or third-party software such as Microsoft's business intelligence platform, PowerBI.

Using third-party tools has advantages over creating custom software for visual presentation (F/R 3.2.1). These include:

- Third-party (business intelligence) BI platforms provide all the necessary analytical functionality, making it easy to create visual representations with minimal programming,
- the advanced tools on these platforms offer various approaches to visualise user-based activity logs for log comparison (F/R 3.2.2) and
- maintenance and editing of these representations are usually trouble-free, with ample support and guides for developers to update custom visual presentations.

There are, however, disadvantages to using third-party tools. These include:

- Third-party BI platforms are likely to require a subscription, which can be costly for a company licence,
- additional courses may be necessary to utilise the capabilities of these platforms fully and
- additional functionality may be required for log extraction (F/R 3.1.3) to import data into the platform.

With the drawbacks listed above, third-party business intelligence platforms are the better visual presentation tools for the system utilisation analysis if the tools are available for use.

2.3.4 Maintenance prioritisation

Maintenance prioritisation is performed from logarithmic analysis. Maintenance priority functional requirements (F/R 4) are shown in Table 2.12.

Table 2.12: *Maintenance prioritising functional requirements (F/R 4)*

Req. ID	Name	Description
F/R 4.1	System utilisation analysis categories	The system utilisation analysis categories are needed to complete the logging analysis. These categories will provide the data necessary to recommend prioritising software maintenance using these log analysis metrics.
F/R 4.2	Maintenance prioritisation factor	The maintenance factor measures the amount of maintenance required for a given software system. This is used to prioritise the software maintenance efforts of the software developers using a scoring system.

System utilisation analysis categories

The system utilisation analysis categories functional requirements (F/R 4.1) are used to categorise the log data for maintenance prioritisation metrics. Table 2.13 outlines the functional requirements of the system utilisation analysis categories (F/R 4.1).

Table 2.13: *System utilisation analysis categories functional requirements (F/R 4.1)*

Req. ID	Requirement name	Description
F/R 4.1.1	Users	Users of software systems can be placed in different categories according to who uses the software. This can be both the customer users and the employees using the software. Using the activities of the customer users will provide the data for the development team to need their resources.
F/R 4.1.2	User activity types	User activity types in Table 2.4 can be used as a category to compare different types of user-based activities and use a subcategory for categories such as different users who can use the system (F/R 4.1.1).
F/R 4.1.3	Systems or subsystems	The request origin (F/R 1.3.5) of user-based activities can be classified to compare different subsystems and controllers.
F/R 4.1.4	Miscellaneous categories	This user-based activity category will use the metadata attribute (F/R 1.3.7) of Table 2.5. The other fields not set as main categories can also be placed in this category because they can take multiple forms.

The categories in Table 2.13 allow the log data to be placed in different categories for comparison. A software maintenance prioritising model can be made using the comparisons of these categories.

Maintenance prioritisation factor

The maintenance prioritisation factor (F/R 4.2) uses various parameters created from the logs obtained. This prioritisation factor will need to indicate, for any given subsystem's software maintenance prioritisation, using a consistent method to determine the maintenance prioritisation factor. Table 2.14 outlines the functional requirement for the maintenance prioritisation factor (F/R 4.2).

Table 2.14: *Maintenance prioritisation factor functional requirements (F/R 4.2)*

Req. ID	Requirement name	Description
F/R 4.2.1	Users	The total number of active users in the subsystems should be a parameter.
F/R 4.2.2	Priority	Priority is a precalculated parameter that indicates the importance of a subsystem. This can be calculated or defined for each subsystem.
F/R 4.2.3	Activities	The total number of user activities for a subsystem must be used as an input parameter for the software maintenance factor.
F/R 4.2.4	Software maintenance factor	The software maintenance factor measures the importance of a certain software system. This is calculated using F/R 4.2.1, F/R 4.2.2, and F/R 4.2.3.

A method can be used to determine the prioritisation of each subsystem's software maintenance using the functional requirements in Table 2.14. The defined parameters provide a basic guideline on the input and output parameters the method needs for the calculations.

2.4 Design

The design of the log mechanism and log analysis will be defined in this section, as well as the flow diagrams of the logging mechanism and log analysis.

2.4.1 Web application architecture

To determine the types of user activity for a web application, the architecture of the Web application will be a factor in the logging mechanism. Web applications consist mostly of HTML, JavaScript, and CSS programming languages. The Model-View-Controller (MVC) architecture is mostly used for web-based applications using that programming language [60]. The MVC architecture in Figure 2.4 comprises of three basic parts [60]:

- *Model:* The representation of the records in the database, which also interacts with the database through a database access layer or service manipulating the data using the CRUD operations:
  - *create* operation that adds new data,
  - *read* operation that gets the data from the database,
  - *update* operation that modifies the existing data, and
  - *delete* operation that removes data.
- *Controller:* Operates both the *View* and *Model* and serves as the connection between the user and the system by controlling the data flow of the *View* and *Model*. *View*.
- *View:* Indicates the results of the data in the *Model* and enables the user to manipulate the data. The user will only interact with this part of the web application.

Figure 2.4 illustrates the equivalent representation of the MVC architecture of Figure 2.2 where the data flow of the MVC architecture is shown. The user interacts with the web application through their browser, which sends a *HTTP requests* to the *Controller* and receives a *HTTP response*[2] from the *View*. The *Controller* will request and return the data to the *Model*, which interacts with the database access layer or service to do the *create*, *update*, and *delete* operations.

To classify any interaction between the user (F/R 1) and the server (F/R 2) to meet the functional requirements of Table 2.3, only the *HTTP request* is used for the logging points in Section 2.3.2 because:

[2] An **HTTP response** is made by a server to a client. The response aims to provide the client with the resource it requested, inform the client that the action it requested has been carried out, or else inform the client that an error occurred in processing its request. Refer to the source: IBM, "HTTP responses", IBM, Available: `https://www.ibm.com/docs/en/cics-ts/5.2?topic=protocol-http-responses` (visited on 2023-07-24)

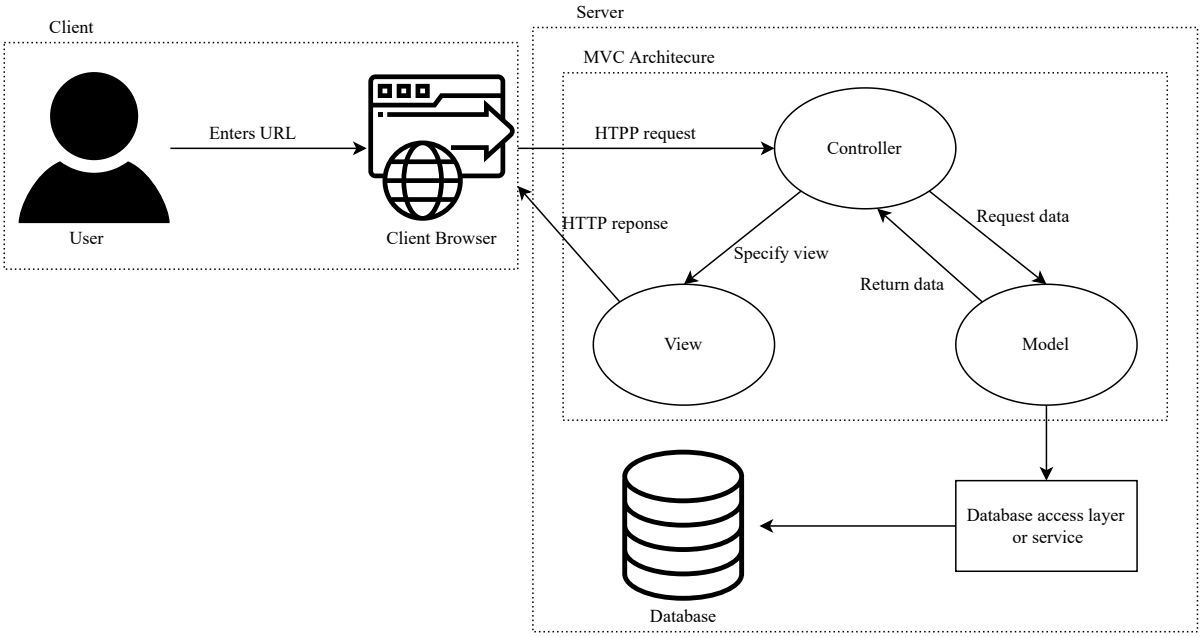

Figure 2.4: *MVC architecture for most web-based applications [61]*

- It meets F/R 1.1.1 and F/R 1.1.1 when the user interacts with *View* to modify the data that is sent back through the *HTTP request* to process the data on the *Controller*,
- The user activity types can be assigned for different scenarios the user triggers when the request is sent, and
- any additional metadata can be sent with the *request header*[3] of the *HTTP request*. This will reduce the overhead the logging mechanism adds by not sending an additional *HTTP request* back to the server each time a user-based activity has been identified.

[3] A **request header** is an HTTP header that can be used in an HTTP request to provide information about the request context so that the server can tailor the response. For example, the Accept-∗ headers indicate the allowed and preferred formats of the response. Refer to the source: MDN Web Docs, "Request header", MDN Web Docs, Available: `https://developer.mozilla.org/en-US/docs/Glossary/Request_header` (visited on 2023-07-24)

2.4.2 Server-side logging point

Section 2.3.2 discusses the functional requirements for the logging point that must be met to create a suitable logging point for a software system to track user-based activities. The *HTTP request* will call a function in the system, subsystem, or web page to execute the user's actions. This *request* information can then be obtained and parsed onto the logging point.

The logging attribute can be created as a centralised code segment that all the software system's components can execute before executing the targeted software system in web applications. These centralised code systems comprise:

- In most software frameworks, the *HTTP request* data can be extracted from the inbuilt *HTTP request* models to obtain the custom request headers set on the client side. There should be at least one common global location in the software where a single log point can be called for every *HTTP* function to execute the log point first before continuing with the rest of the main targeted function. The rest of the log attributes can also be obtained during the execution run time:
  - **Absolute Uniform Resource Identifier (URI) path:** The string containing the absolute URI path of the currently active system, subsystem, or web page.
  - **Absolute request Uniform Resource Locator (URL):** The requested URL contains the target system, subsystem, or web page name and function that the request needs to be executed.
  - **Action parameters:** During the execution time, some of the parameters, which are the request parameters sent with the *HTTP request* from the client device, can be obtained.
- In other older web applications created with programming languages such as `PHP`, a more direct approach must be taken when accessing the requested data. In this case, multiple logging points that call the logging points' main code segment are used to capture the attributes and store the log in a database. The parameters may need to be extracted before parsing them into the main logging point code segment.

As long as the logging attributes and the *HTTP request* headers are obtainable, the logging mechanism can be created on the server side to extract and process the data. The defined cases can resolve the activity type; for example, if the request calls the `Index` function of the system, subsystem or webpage, it can be identified as the web page accessed user activity type (F/R 1.2.1) of Table 2.4.

If the user-based event uses the system, subsystem, web page, or functions that modify the session, it can be classified as a session-changed event (F/R 1.2.3). Subsequently, the

remaining user-based activity events must be tested to see if they meet specific criteria defined for the general activity types. Suppose a user-based event fails all three types of classification. In that case, the event is likely not user-generated or comes from a *HTTP request* executed after the initial first request. In such cases, the last HTML element ID that triggered the event should not be listed as a clicked element in JavaScript.

2.4.3 Client functional requirements interaction

Figure 2.5 illustrates the user interacting with the user interface to trigger a user-based activity event to be logged later for the client's functional requirements. The process starts with the user interacting with the user interface. The default activity type is set to general activity (F/R 1.2.3) until it is further processed later in the logging mechanism.

If the activity has additional metadata, such as other request parameters, it will be logged by adding it to the search for it in the completion of the *HTTP request* operation. The additional metadata can also be captured from the client side in this stage, such as the element that the user clicked to initiate the event. The captured metadata are then placed in a custom request header, and the *HTTP request* continues its normal operations and sends the data back to the server.

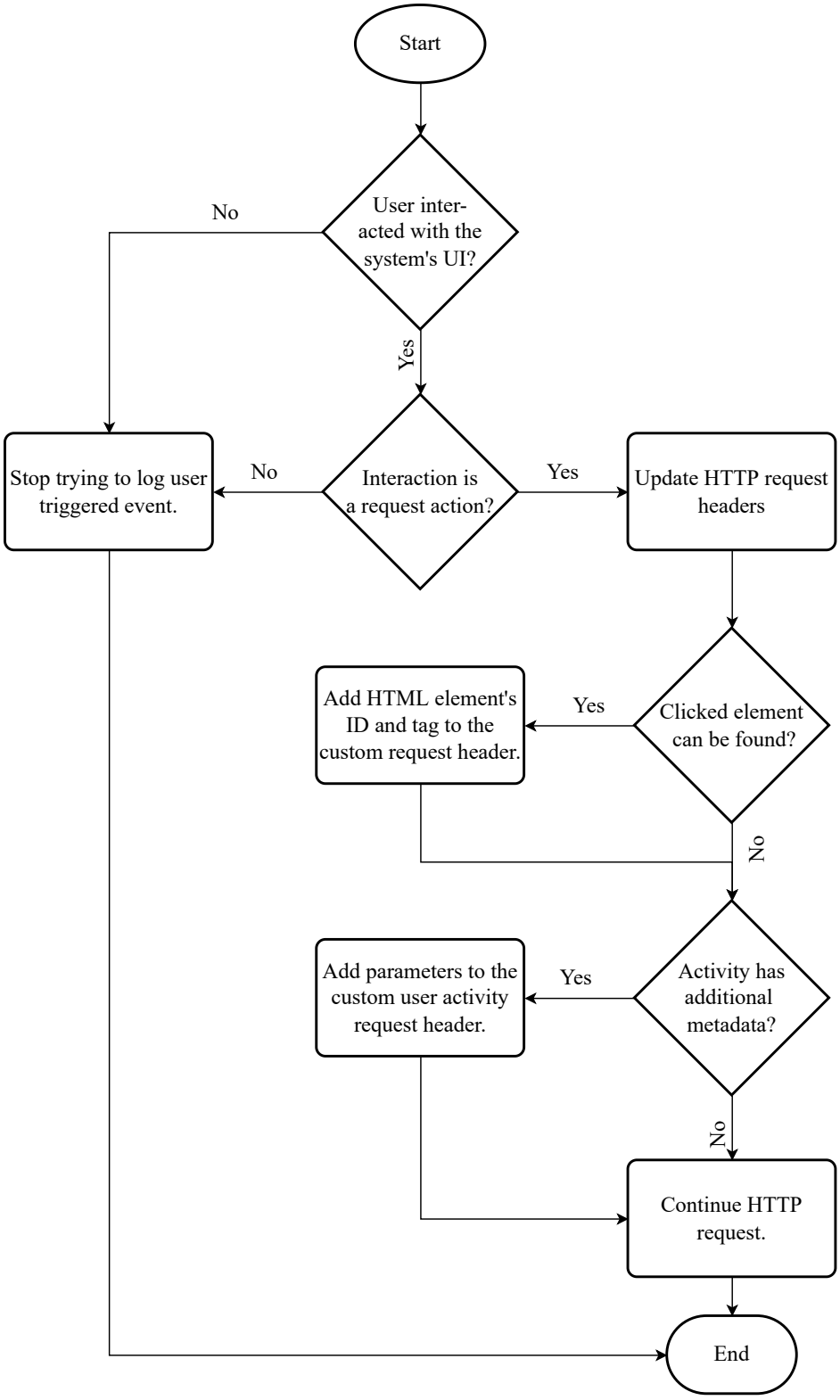

Figure 2.5: *User-based activity log classification flow diagram*

2.4.4 Server-side log parsing

Figure 2.6 illustrates the server-side log parsing of the possible user-based activity events obtained for the software system.

The defined globally-set logging point will start the user-based activity process before the targeted process Figure 2.6 is executed. At this stage, if something goes wrong with the logging during the execution of this filter, it should be abandoned, and the software system should be left to continue to ensure that it does not interfere with the software system's operations (F/R 1.3.4 of Table 2.7).

In the case of the request method `NULL` or empty due to errors such as incorrect parameter types for the targeted procedure in the controller, the logging point should stop attempting to log the user-based log. The issue would most likely appear as a runtime error, and any user-based activity log procedures will likely fail due to incomplete data or render the logs incomplete or inconsistent (F/R 1.3.3 of Table 2.7).

If the captured user-activity log contains any parameters, it should be checked for any session-related parameters or any other potential user data that should be removed from the metadata to prevent any personal information from being accessed by anyone, not the user account owner. If the user-activity contains no request parameters, `ElementsInfo` should be set to `NULL`.

In Listing 2.2 is the `JSON` data of `ElementInfo`.

```
1  { "ElementTagName" : "button",
2    "ElementID" : "submitButton",
3    "ElementDataKey": "submit-control"
4  }
```

Listing 2.2: *Element properties JSON*

Each of the properties in Listing 2.2 consist of:

- `ElementTagName`, is the HTML element's tag name, which is one of the defined accepted tag names, such as `button`, `label`, and `td` etc.,
- `ElementID` is the identification of the element if it has been assigned to the element and can be obtained on the client side and
- `ElementDataKey` is additional captured data attributes that expand on the element's identity if it is a custom-made HTML element control. Some software systems may have other custom-created HTML elements, which can also trigger a user-based activity. Other miscellaneous elements, such as a `label`, may not be normal input controls.

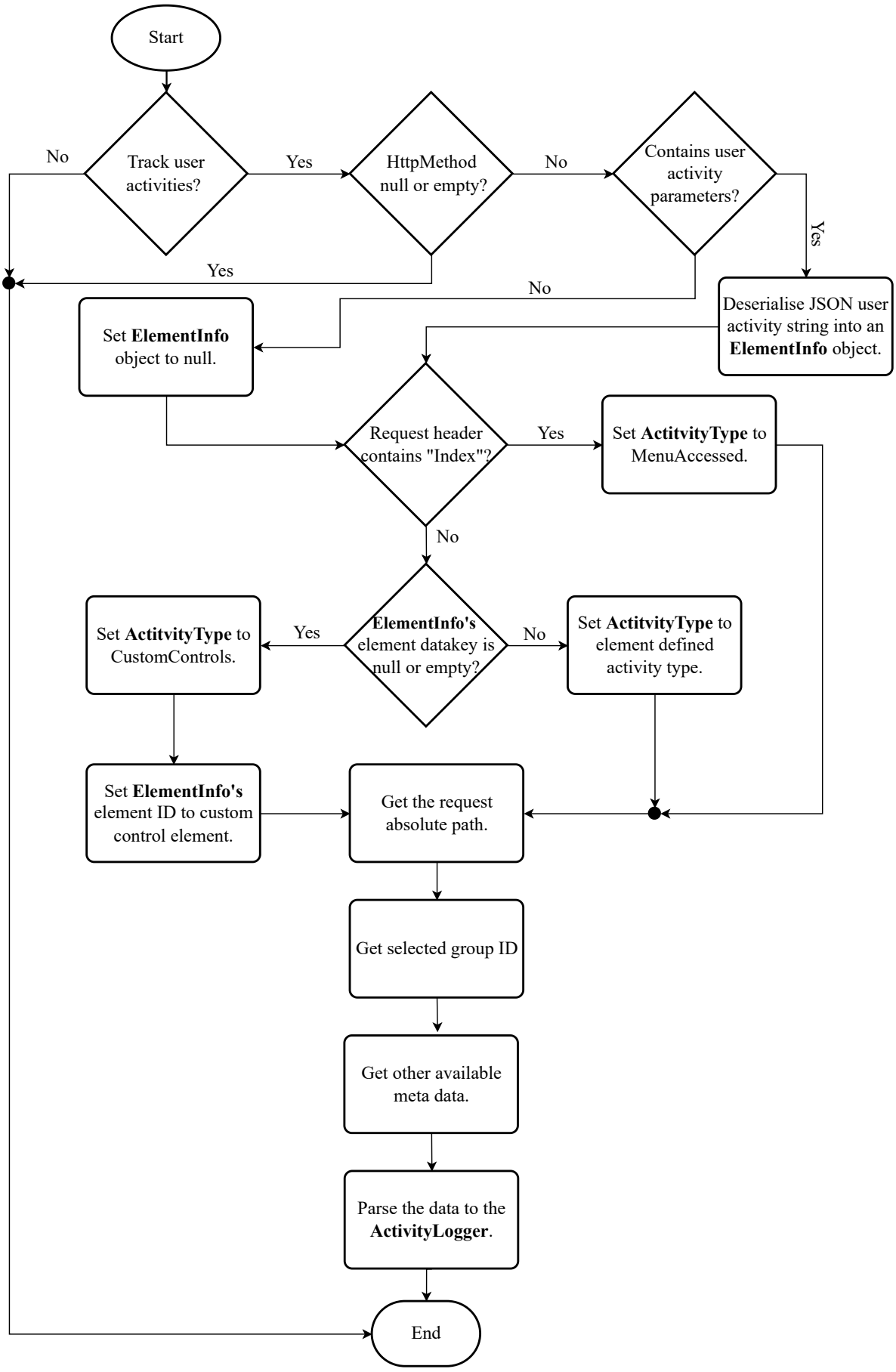

Figure 2.6: *Server-side log parsing flow diagram*

Suppose the `request header` contains the `Index` keyword, which is the first procedure that needs to be executed for a web page being accessed. The activity can be classified as accessing a system, subsystem, or web page. This activity type is the first user activity type at this point of the log parsing before it is processed again to another user activity type.

If the request does not contain the `Index` keyword, the process will continue to the next operation that will check if the `ElementInfo`'s `ElementDataKey` is either a null or empty value. If any data is available, the activity type can be set to the custom control-defined activity type or custom control to represent the custom-made elements. The `ElementID` is set to the custom control element or the defined custom control's identification.

If the `ElementInfo`'s `ElementDataKey` is null or an empty value, the user activity type is set to the element's defined activity type. After the activity type is resolved, the request origin of the user-based activity is obtained by getting the request's absolute path.

After the request origin has been obtained, other relevant session information, such as the group that represents a specific entity data, can be obtained, as well as the user's identification and other relevant metadata that is available at this stage to complete the log attributes that need to be captured from Table 2.5 to complete the user-based activity log.

The data are parsed to the activity logger that will write the log into a database if the log was successfully obtained. This will end the logging process until a new user-based event log is ready to be processed and stored in the database.

2.4.5 Storage of event logs

The log attributes in Table 2.8 will have foreign key references to other tables in the database for other tables, as seen in Figure 2.7.

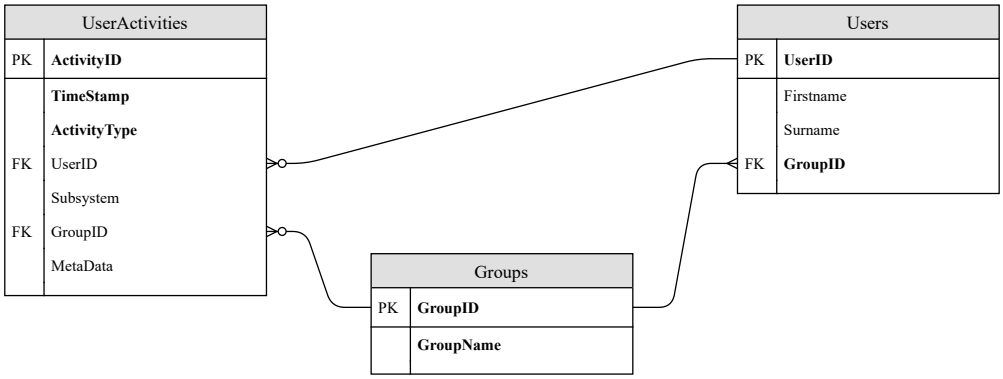

Figure 2.7: *ERD of the user activities*

Figure 2.7 illustrates an Entity Relationship Diagram (ERD) that describes the relationship of the table created to store the log attributes with other relevant tables. This enables different fields of the other tables to categorise the logs in the system utilisation analysis. Figure 2.8 illustrates the flow diagram of the database interaction to store the event log data.

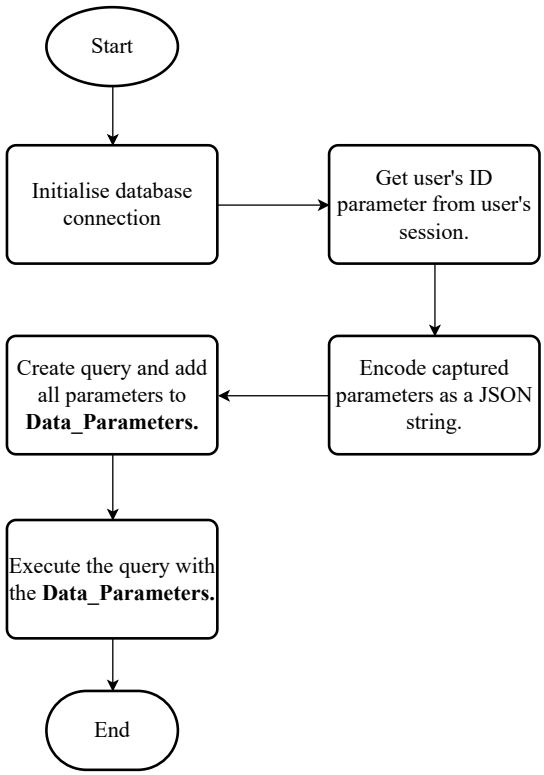

Figure 2.8: *Database interaction flow diagram*

Figure 2.8 illustrates how the connection to the database is established. The user's identity is obtained from the active session; the obtained metadata are then converted into a JavaScript

Object Notation (JSON) string for storing. The final query is built and executed to store in the relational database.

2.4.6 Maintenance prioritisation

The system utilisation analysis aims to provide maintenance recommendations to developers to improve their maintenance efforts by:

- Prioritising maintenance efforts on systems that are more frequently used and
- decommissioning new systems. User-based activities provide a quantitative reason why specific systems can be decommissioned due to users' inactivity.

This can be done by normalising the user-based activity data. Normalising data is a common practise in data analysis and modelling for several reasons [62]:

- **Comparability:** Normalising data allows fair comparisons between variables or subsystems. Bringing different metrics to a standard scale makes assessing their relative importance and impact easier.
- **Eliminating scale effects:** Normalisation removes the influence of the measurement scale, ensuring that the relative relationships between variables are preserved without being skewed by their original scales.
- **Improved convergence:** In some analytical techniques, such as optimisation algorithms, normalising data can lead to faster convergence and more stable results.

Using normalisation the maintenance prioritisation uses Equation (2.1) created from the functional requirements for maintenance prioritisation (F/R 4.2):

$$M_{PF} = A_N \times P_N \tag{2.1}$$

where:

- M_{PF} is the maintenance priority factor,
- A_N is the normalised activity for the total logs obtained per subsystem and
- P_N is the normalised priority factor for the subsystem.

Normalised user activities per subsystem are described by Equation (2.2):

$$A_N = \frac{A_X - A_{Min}}{A_{Max} - A_{Min}}, \tag{2.2}$$

where:

- A_X is the total obtained user activities for the specified subsystem,
- A_{Min} is the total minimum user activities for a subsystem, and
- A_{Max} is a subsystem's total maximum user activities.

The normalised priority factor P_N is described by Equation (2.3):

$$P_N = \frac{P_X - P_{Min}}{P_{Max} - P_{Min}}, \tag{2.3}$$

- P_X is the total obtained users that have access to the subsystem,
- P_{Min} is the total minimum number of users that have access to any subsystem,
- P_{Max} is the maximum number of users accessing any subsystem.

A normalised value for the priority is used because there is no predefined way to describe the importance of any subsystem. Figure 2.9 illustrates the log-analysis flow diagram of implementing the maintenance prioritisation for a software system.

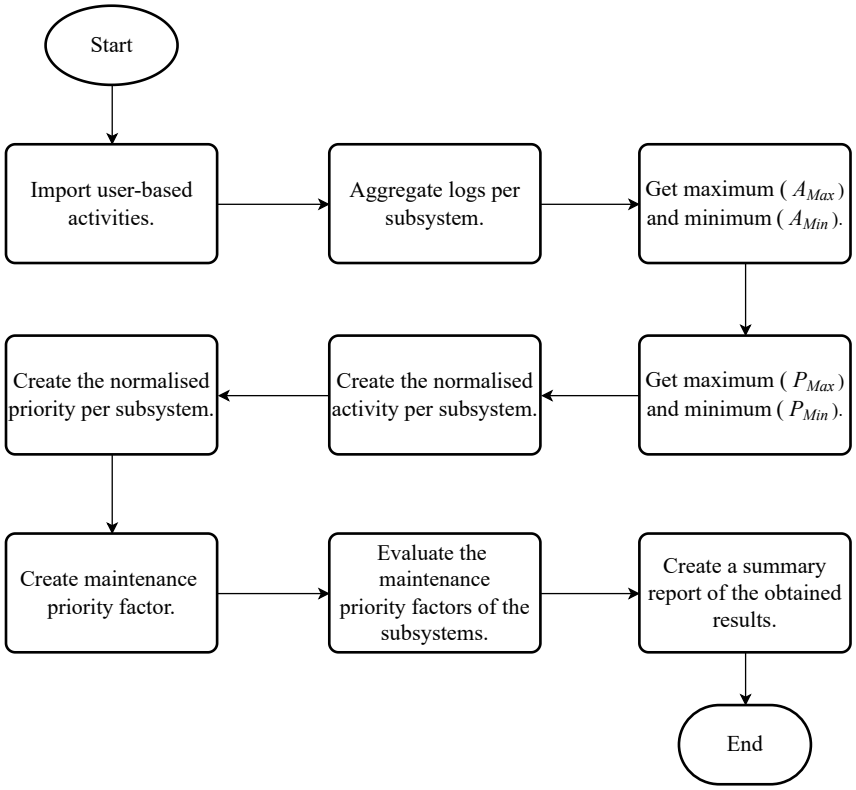

Figure 2.9: *Maintenance prioritisation flow diagram*

As seen in Figure 2.9, the user-based activity logs are imported into the log analysis tool from the relational database. The logs are aggregated into unique subsystems. The total number of user actions is counted, as well as the total number of unique users associated with the captured logs. The maximum and minimum total user activities (A_{Max} and A_{Min}) and users (P_{Max} and P_{Min}) are obtained for the software system.

The normalised activity (A_N) for each subsystem is created from the obtained activities. Normalised priority (P_N) is created from the maximum and minimum priority obtained. The maintenance priority factor is created from each system's normalised parameters and evaluated. A summary report is created for the results obtained from the log analysis using the maintenance prioritisation method.

2.5 Conclusion

Section 2.3 discussed the defined functional requirements for a logging mechanism for the system utilisation analysis outlined in Sections 2.3.3 and 2.3.4. The user activity types defined in Table 2.3 are the basis for the log attributes that must be logged to create a user-based log.

The log points capture the logs and send them to the server logging point, where they are processed and stored in writing in a database.

The logs are extracted into a visual presentation that enables maintenance improvement suggestions based on the use analysis in Sections 2.3.3 and 2.3.4.

Chapter 3

Results

3.1 Introduction

In Section 2.1, the methodology for a web-based software system was created. As described Section 2.1, users can have multiple systems and subsystems linked to their accounts. The implementation is done for various software systems in Section 2.1. The system implementation will focus on developing the logging mechanism, and the critical analysis will prioritise software maintenance using a utilisation analysis.

3.2 Implementation

In this section, the implementation of the development of the solution will be discussed using a verification test system. The test system is created in a `ASP.NET Core Web SDK` software environment.

3.2.1 User activity types

Figure 2.5 illustrates the user-based clarification flow diagram that utilises Table 2.3 to identify and capture some of the log attributes. Event logs should consist of multiple cases (F/R 1.1.2) that are the primary identifiers for the event log, which are user activity types.

Using the basic operations of the system and what users interact with, user activity types can be made for each software system. For the test system, the basic use will:

- monitor resource usage dynamic dashboards, and
- generate `PDF` reports of the displayed data.

Reporting and monitoring are important for this test system; therefore, the user activity types must focus on these. To satisfy the F/R 1.1.2, F/R 1.1.5, and F/R 1.1.6 user activity types, it is defined in Table 3.1.

Table 3.1 doesn't contain any session changes (F/R 1.2.3) user types. These activities are only triggered when the user logs into their system or terminates their session by pressing the log-out button.

Table 3.1: *Test user activity types*

Activity	Functional requirement	Description
`SystemAccess`	F/R 1.2.1	This activity type detects when a user has navigated to a certain subsystem.
`General`	F/R 1.2.3	This general activity type is for all other activities that the user initiates that send *HTTP requests* back to the server.
`ReportExport`	F/R 1.2.3	This system's main function is for reporting purposes. Separating this type of activity into its category captures all report generation activities that the user has initiated.

3.2.2 Log attributes

Using the functional requirements discussed in Section 2.3.1, data columns are made in this section for the log attributes of the user-based event. These log attributes for a structured database are defined in Table 3.2.

Table 3.2: *Logging attributes*

Column name	Requirement ID	Description
`ID`	F/R 1.3.1	User based activity primary identifier.
`TimeStamp`	F/R 1.3.2	Date timestamp when the activity occurred.
`ActivityType`	F/R 1.3.3	Activity type of the log event.
`UserId`	F/R 1.3.4	The user identification number associated with the log event.
`SystemId`	F/R 1.3.5	System where the activity occurred.
`SubsystemId`	F/R 1.3.5	Subsystem where the activity occurred.
`MetaData`	F/R 1.3.6	Metadata captured from the *HTTP request*.
`ClientId`	F/R 1.3.7	Additional identifiers for the log event. In this case, different configurations of the same system for a specific client.

The data columns in Table 3.2 are used in a structured database. A *MySQL* database is used to implement the relational database. This database has preexisting tables that expand on other data such as the `UserId`, `SystemId` and `SubSystemId`. For the `MetaData`, the `JSON` string is similar to Listing 3.1.

```
{
  "RequestOrigin": "/System/Subsystem1/GetData",
  "RequestElementID": "ButtonSaveCsv",
  "RequestParameters": {
    "tagIds": [
      "6284",
      "20320"
    ],
    "toDate": "2020-04-06",
    "groupId": 2,
    "fromDate": "2020-03-30"
  }
}
```

Listing 3.1: *Metadata JSON*

In Listing 3.1, the main `JSON` parameters capture the following additional data for the user-based event:

- `RequestOrigin` is the complete file path of the subsystem used and the functions required to fulfil the *HTTP request*. Some subsystems consist of multiple individual files, which trace the origin of the user-based activity accessed from the *HTTP request* function.
- `RequestElementID` is the last *HTTP* element identification that the user interacted with that initiated the user-based activity.
- `RequestParameters` is the request parameters that are sent with the *HTTP request*. Any sensitive user data are ignored by adding flags to certain subsystems or individual functions to obtain the request's parameters.

Obtaining the element of the user-based event

In Section 2.4.1, the user-based activity event used a *HTTP request* to send to the server when the user interacts with an *HTML element*. For the functional requirements activity type (F/R 1.5.3) and metadata (F/R 1.5.6) in Table 2.5, the *HTML element* needs to be obtained to obtain the element's tag and identification text.

This cannot be easy to obtain due to *bubbling*[1] that may occur when searching for the element with which the user specifically interacted. Figure 3.1 illustrates the propagation of the bubbling event.

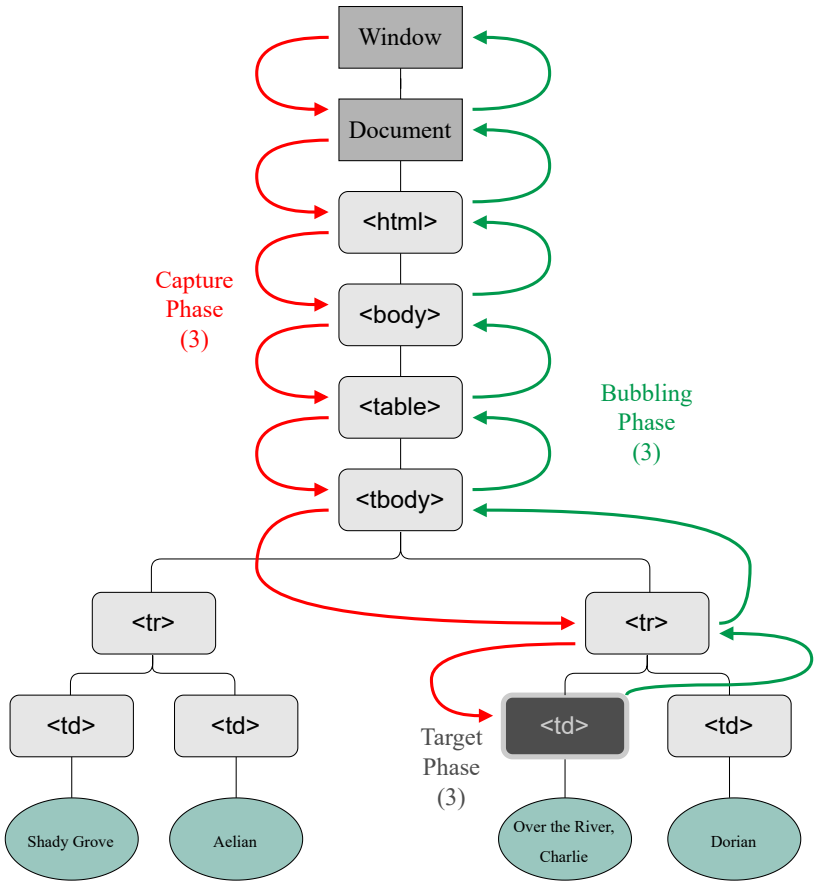

Figure 3.1: *JavaScript event propagation*

Figure 3.1 illustrates the example of an event propagation of a child element that executes a Document Object Model (DOM) event when clicked on. Event propagation consists of three phases:

- *Capturing phase:* The event propagates downward to the target element with which the user interacts.
- *Target phase:* The event reaches the targeted element to execute the DOM event.
- *Bubbling phase:* The event bubbles up from the target element.

Capturing the targeted element may be difficult because some web pages may have more complex HTML, which can cause event propagation to fail to obtain the correct element information that the user interacts with. In such cases, obtaining the target element by identifying the last known element the user hovered over on the user interface is more accurate

[1] **Bubbling** is when an event happens on an element. It first runs the handlers on it, its parent, then on other ancestors. Refer to the source: JavaScript.Info, "Bubbling and capturing", JavaScript.Info, Available: `https://javascript.info/bubbling-and-capturing` (visited on 2023-07-24)

since another DOM event may have started during the initial element's event.

Figure 3.2 illustrates the flow diagram to capture the element with which the user interacted for the user-based activity log. This code segment will be initiated during the `beforeSend` operation of the *AJAX request* to filter HTML elements by predefined allowed elements. Filtering the element tag names ensures that unwanted, more complex, essential elements not expected to be the event's initiator will be excluded.

If the web location has already changed or no element exists, the page's contents may have changed during the event propagation. Therefore, the last known component that the user hovered over must be used because it is most likely the element that the user interacted with. This approach ensures that a component has always been detected and synchronised with the request header in most UI changes.

3.2.3 Logging points

Table 2.7 outlines the functional requirements for logging points in the software system. It is crucial to maintain consistency when capturing logs. Placing the captured logs globally for all HTTP requests will ensure consistency. Filters can be used during the test system, as it is an `ASP.NET Core Web SDK`. Filters can be initialised during the software system startup phase.

For other web-based systems, the central point where *HTTP requests* are processed should be used to place single or multiple logging points. In other cases, adding it for smaller groups of subsystems is also viable if the logs can be consistently captured when requests come through.

The filter of the test system is the primary component of the logging mechanism. Log attributes and metadata can be captured log attributes and metadata on the client side using server-side parsing, as shown in Figure 2.6.

Section 3.2.2 discussed that some of the logs may contain sensitive data that should not be logged, especially the metadata request parameters. Adding a flag to indicate that specific subsystem requests must be ignored for event logging should be included.

This may exclude specific parameters from being saved into the database. It is an attribute added to specific controllers to ignore the event logs obtained and terminate the logging process.

The logging process in Figure 3.3 is placed on the client-side filter, which runs before the rest of the request is handled. It is essential to check whether the request has an action parameter to ensure that the function being executed is called by the request and not by the internal system.

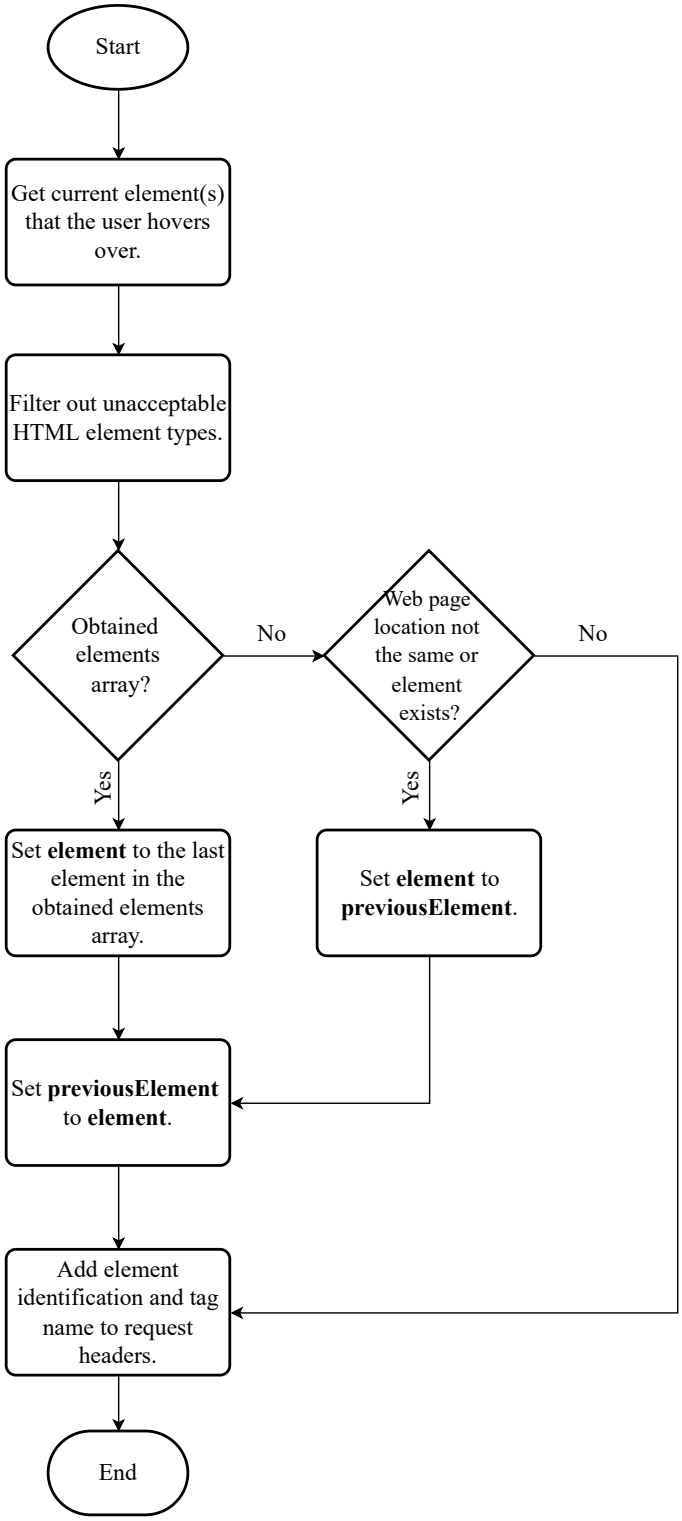

Figure 3.2: *HTML element capturing flow diagram*

If the request has a report action parameter, the user activity should be set to either `Activity1` or `Activity2`. The rest of the request parameters should be obtained and formatted as a `JSON` string to be stored in the database.

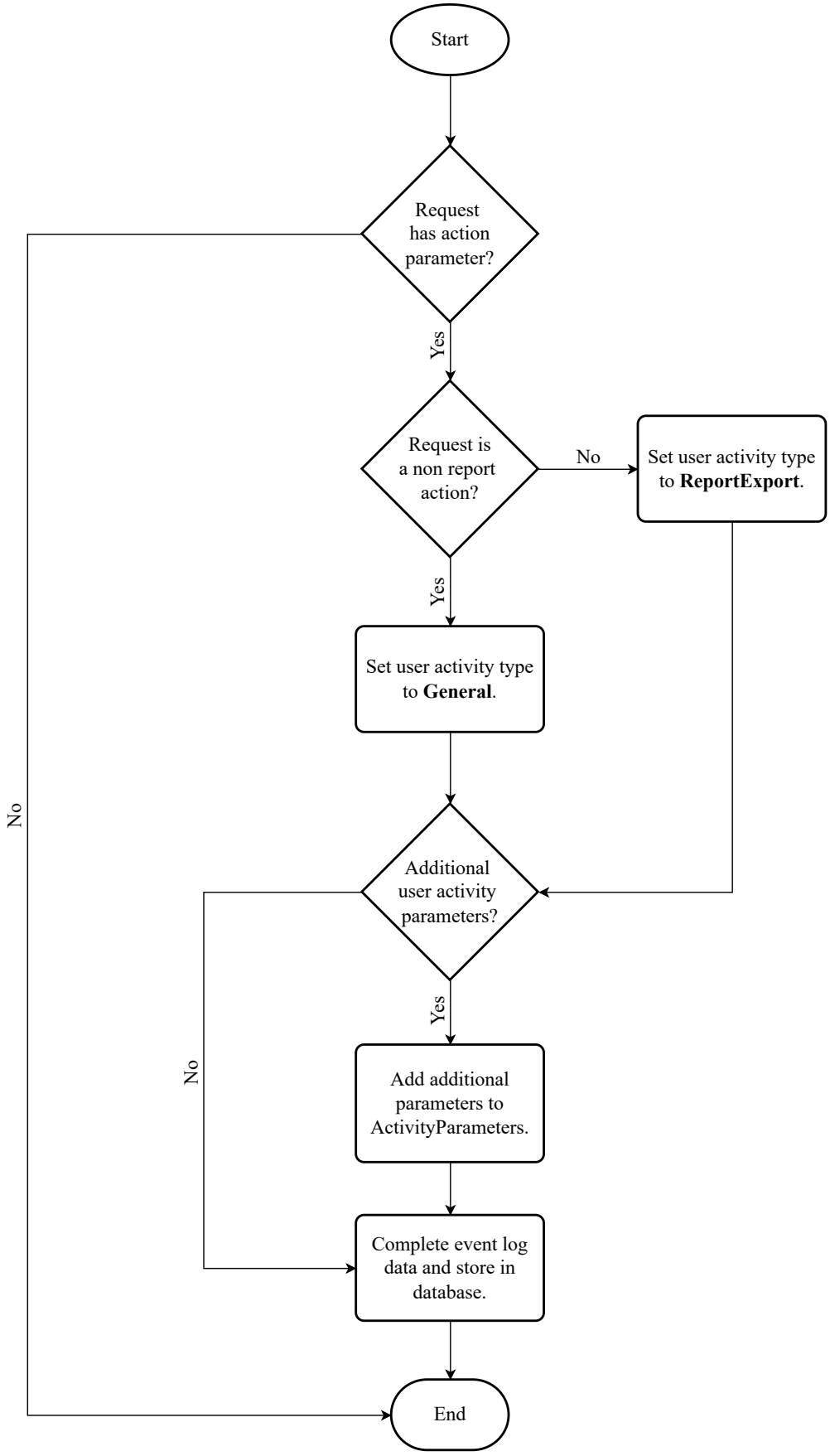

Figure 3.3: *Logging point operation for test system*

3.2.4 Log analysis

In Sections 2.3.3 and 2.3.4, the functional requirements for log analysis are defined. For the implementation of log analysis, a custom log analysis tool is created to:

- Visually present user-based event logs through the log analysis tool as outlined in Table 2.11.
- Filter user-based event logs using different criteria described in Table 2.13.
- Analyse logs for maintenance prioritisation as shown in Table 2.12.

3.3 Verification

The log analysis tool will be used to verify the implementation of the logging mechanism on the test system. The log analysis tool is created in a `.NET Framework` software environment and uses a `MySQL` database to store log events.

3.3.1 Log attributes

A sample of the captured log attributes of Table 3.2 is shown in Figure 3.4.

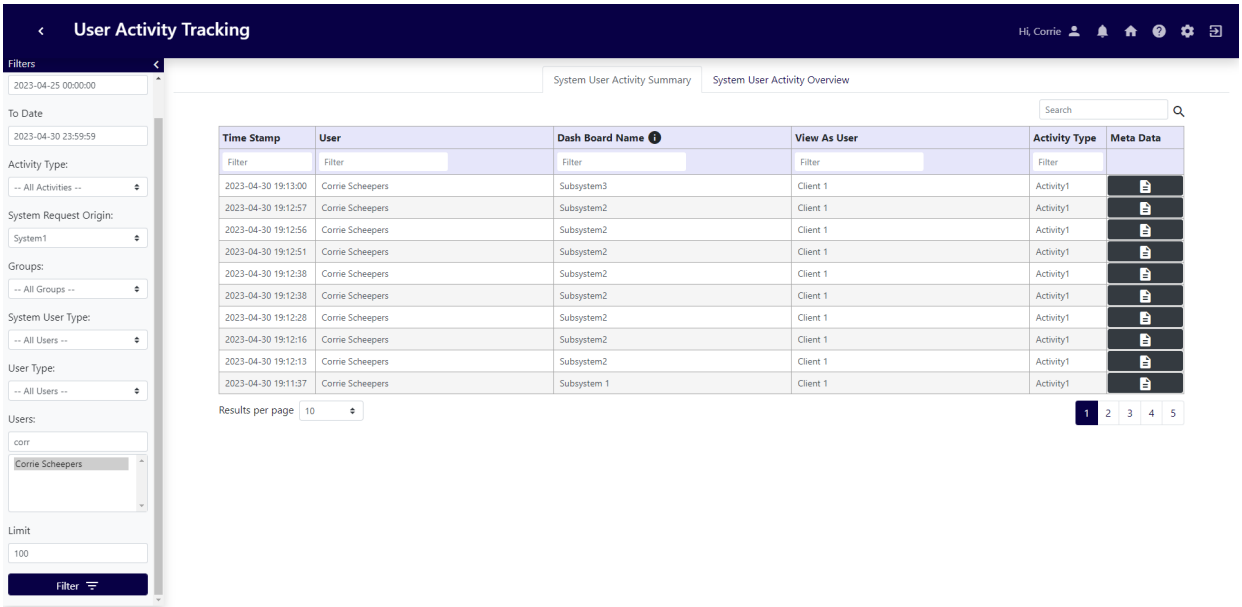

Figure 3.4: *Interactive user activity viewer*

Figure 3.4 illustrates that the required log attributes defined in Table 3.2 are being tracked for the test systems. The user interface created to display the logs is designed to be more understandable to users who analyse the logs. The metadata is displayed in a `JSON` format as seen in Figure 3.5.

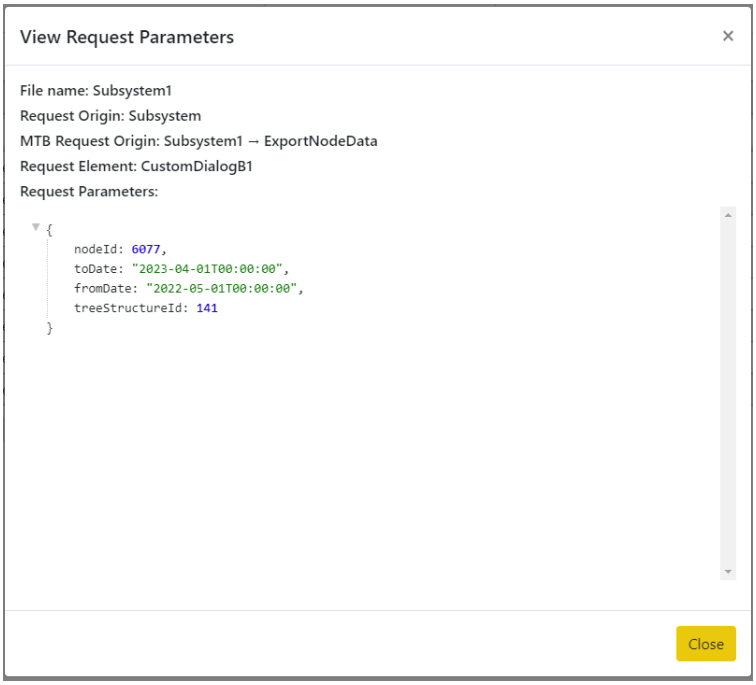

Figure 3.5: *JSON test request parameter data*

Figure 3.5 includes additional parameters described in Listing 3.1. `RequestElementID` is obtained using the element capture method described in Figure 3.2, while the other metadata parameters are captured using the built-in methods available in `C#`.

3.3.2 Log analysis

The log analysis of the logs obtained in Figure 3.4 is done in the same interactive dashboard. Figure 3.6 compares the logs obtained for the subsystems.

Figure 3.6: *Interactive user activity viewer*

The log analysis, as seen in Figure 3.6, illustrates the subsystems' total recorded user activity logs that are compared. Depending on the required logging attributes, different categorical comparisons can be made from the obtained user-based event logs.

By comparing the categories described in Table 2.13, different categories can be compared, as shown in Figure 3.6.

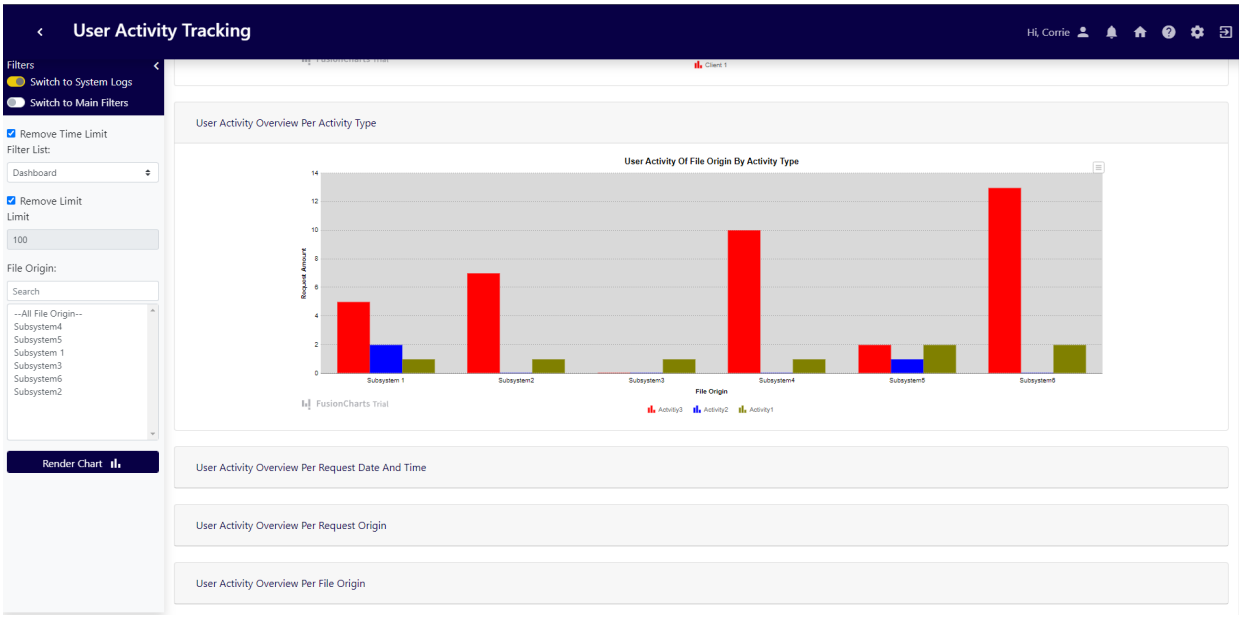

Figure 3.7: *Interactive user activity log analysis*

Figure 3.7 compares user activity types. For these subsystems, `Activity3`, which is the `General` user activity type, is the most recorded user activity type. This type of activity is expected to be the most prominent for this test system.

Log quality

Using the functional requirements for the log quality (F/R 3.1), the log quality of the test system is evaluated in Table 3.3.

Table 3.3: *Logging quality assessment of the test system*

Req. ID	Description	Achieved	Comments
F/R 3.1.1	Log availability	✓	In the implementation of a logging mechanism for the test system, the log points were: • *Locally complete* since all the log attributes were available during the capturing phase of the event log. • *Globally complete* because defined user activity types were captured for the test system. The logging points captured the expected logs as defined in Figure 3.3.
F/R 3.1.2	Log completeness	✓	All required log attributes were obtained during the log capture process of the log point. No post-logging corrections were needed to fix any of the log attributes or to fill in missing log attributes.
F/R 3.1.3	Log extraction	✓	As previously stated all logs were complete. Any additional information extracted from the database is to make the obtained more readable for humans by using the foreign key references to other tables as illustrated in Figure 2.7.

Table 3.3 discussed the log quality of the test system's user-based event logs that is on an acceptable level. Log analysis should be more accurate with the concurrently obtained logs captured by the logging mechanism.

3.3.3 Maintenance prioritisation

For the test system, maintenance prioritisation recommendations can be made based on Section 2.3.4. A rating system can be used to classify the most critical software systems that need prioritisation, which can help in software maintenance efforts.

Using Equation (2.1) the maintenance priority factor M_{PF} can be determined for a set of systems S_1, S_2, ..., S_N which has captured user activities per system A_X. The system will also have several users connected to each system P_X. These parameters are set for the test system as outlined in Table 3.4.

Table 3.4: *Data for validating test system*

System (S_X)	Users per system (P_X)	Number of events (A_X)	Expected priority
S_1	226	11	1
S_2	269	5	2
S_3	156	8	3
S_4	155	1	5
S_5	146	13	5
S_6	154	8	4

The activities in Table 3.4 were generated by a single user who navigated and interacted with the system, as shown in Figures 3.6 and 3.7. To compare the effect of the total number of user activities per system, we focus on S_3 to S_6, which have similar numbers of active users who can access the system.

For the test system S, it is expected that S_1 will have the highest maintenance priority factor, given that it has:

- The second highest number of users linked to it. This should increase its normalised active user factor and
- has the second-highest number of observed user events captured.

S_2 should have the second highest expected priority. It has the highest active user count and half the total user activities captured for S_1. S_3 has a much lower user count than the two previous systems. Third, this should place its maintenance priority, as its normalised user count should be similar to S_4 and S_6.

S_4 has the least number of user events but approximately the same number of users linked to it. It should, therefore, have the lowest maintenance factor since it will be zero due to normalised user activity. Similarly,

S_5 will also have the lowest maintenance priority seen in S_4 since it has the lowest number of users connected to it. This will set the normalised priority to zero.

S_6 should have the fourth highest maintenance priority. It will have a similar normalised user activity count as S_3 but lower normalised priority than all the systems except S_5.

Using Equation (2.3) to calculate the normalised priority factor for each subsystem using the number of users that have access to the system as well as; Equation (2.2) to calculate the normalised activities, the results are shown in Table 3.5.

Table 3.5: *Test data*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_1	226	0.6504	11	0.8333	0.5420	1
S_2	269	1.0000	5	0.3333	0.3333	2
S_3	156	0.0813	8	0.5833	0.0474	3
S_6	154	0.0650	8	0.5833	0.0379	4
S_4	155	0.0732	1	0.0000	0.0000	5
S_5	146	0.0000	13	1.0000	0.0000	5

In Table 3.5, P_N for S_1 and S_2 is the highest, as most users have access to them. Furthermore, S_1 and S_5 have the highest A_N rating, indicating that they were the most used systems. As stated, S_1 is expected to require the most maintenance activities. Systems with lower maintenance activities that still have similar active users linked to them have a lower maintenance priority factor.

3.4 Case studies

3.4.1 Case study identification

Three case studies will be used to examine the application of this study thoroughly. All the case studies are web-based applications where users need credentials to log in and have restricted access to some parts of the system. These case studies are identified in Table 3.6.

Table 3.6: *Case studies*

Case study	Software framework	Description
A	`ASP.NET Core Web SDK`	Energy management software system comprising internal and external client users.
B	`Hypertext Preprocessor (PHP)`	An older energy management software system comprising internal and external client users.
C	`ASP.NET Core Web SDK`	Administrative software system used by internal company users. It consists mostly of Case Study A and B configuration tools and is only accessible to internal users.

The case studies in Table 3.6 were selected because they have different use cases and software framework implementations. Case Studies A and B are similar software systems but use older and newer software frameworks. Due to these differences, the logging mechanism must be implemented differently to capture user-based activities.

For the three case studies, the following results will be obtained:

- Defining commonly occurring user-based activity types.
- Defining how the logging point is implemented for the specific subsystem to obtain the user-based event logs.
- Only user activities of the subsystems in the upper quartile recorded in October 2022 are used for each case study. See Appendix B for the results of the remaining full case studies.
- The normalised priority for each subsystem of the main system will be calculated using Equation (2.3) based on active users who have access to the system.
- Normalised activities of each subsystem will be calculated using Equation (2.2).

- The maintenance priority factor will be calculated for each subsystem using Equation (2.1).
- The normalisation of the priority factor is only for users who have access to the subsystem and interact with the subsystem. All other users who do not meet this requirement will be excluded.

3.4.2 Case Study A results

Case Study A is a software system `ASP.NET Core Web SDK`. The software system has three basic user activity types, as shown in Table 3.7.

Table 3.7: *Case Study A activity types*

Activity	Functional requirement	Description
`Dash`	F/R 1.2.1	This activity type detects when a user has navigated to a certain subsystem.
`DetailView`	F/R 1.2.3	This general activity type is for all other activities that the user initiates that send *HTTP requests* back to the server.
`Report`	F/R 1.2.3	The other main function of the system is for reporting purposes.

The logging point is placed in a central location in the software code to capture the types of user activity defined in Table 3.7. Using the action filters available in `ASP.NET Core`, the single logging point can be placed in the software system to capture the user-based event logs.

Additional metadata, such as the HTML element associated with the user-based event, is also captured. Using the HTML event capture method shown in Figure 3.2, element information is captured and stored together with the request parameters in the same format described in Listing 3.1. Figure 3.8 breaks down total user activities captured of the user activity types of Table 3.7.

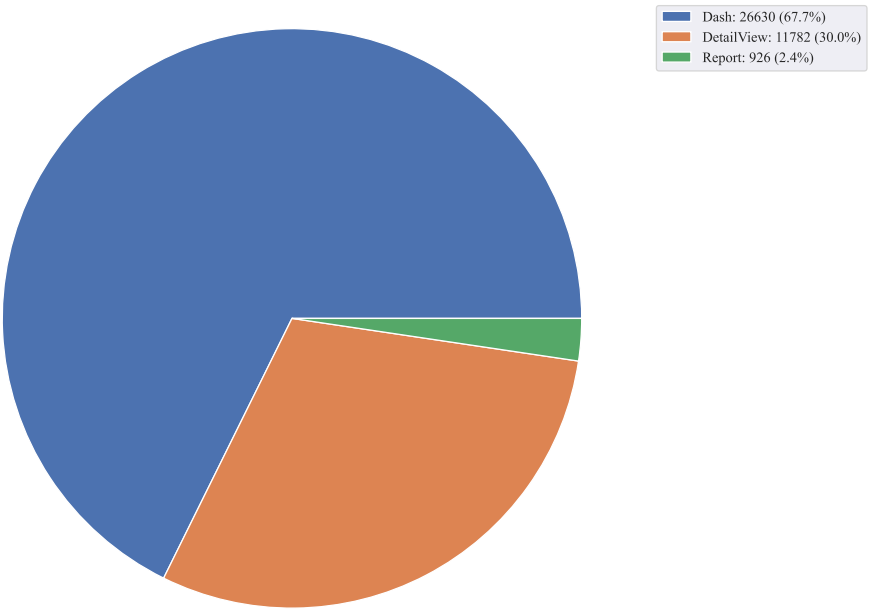

Figure 3.8: *User activity types breakdown of Case Study A*

Figure 3.8 highlights that most of the user activities for Case Study A's subsystems are the Dash user activity type. Most subsystems do not have many inputs for the user to enter data into the system. Figure 3.8 provides information back to the user that causes most of the user activities. Figure 3.9 breaks down each subsystem's utilisation user activity.

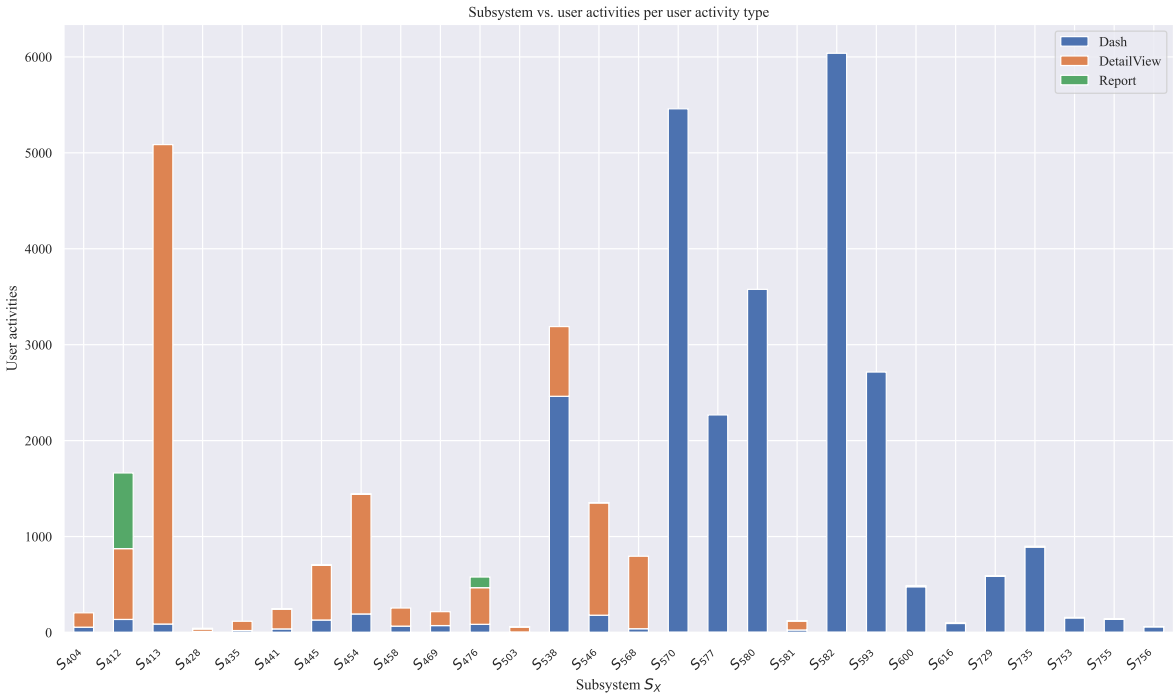

Figure 3.9: *System utilisation breakdown of Case Study A*

In Figure 3.9, each subsystem's total utilisation is further broken down into user activity types defined in Table 3.7. For systems such as S_{413}, most user activities were from actions

the user performed other than accessing the subsystem, which is the DetailView user activity type. This differs from the other top five systems, with most Dash user activity types. Users only viewed the content of the web page for that subsystem. The log quality assessment for Case Study A is discussed in Table 3.8.

Table 3.8: *Logging quality assessment of the test system*

Req. ID	Description	Achieved	Comments
F/R 3.1.1	Log availability	✓	The log availability for Case Study A is: • *Locally complete* since all the log attributes were available during the capturing phase of the event log. • *Globally complete* due to using a central logging mechanism in the main system that can capture user-based events.
F/R 3.1.2	Log completeness	✓	This system uses the same log attribute structure described in Table 3.2 that is captured with the client- and server-side logging points of this case study. All the log attributes formed a complete log without additional post-logging operations to correct or fill in missing attributes.
F/R 3.1.3	Log extraction	✓	As previously stated all the logs were *locally-* and *globallly* complete. This enables log extraction during log analysis.

Case Study A's log quality is satisfactory for the log analysis. The three basic user activity types allow the logging points to make simple operations to classify the user-based activities.

Maintenance prioritisation

The maintenance prioritisation factor for the user activities in the upper quartile of the subsystem for Case Study A is calculated in Table 3.9.

Table 3.9: *Case Study A's upper quartile maintenance performance*

S_X	P_N	A_N	M_{PF}	P_R
S_{538}	1.0000	0.5281	0.5281	1
S_{413}	0.5469	0.8423	0.4606	2
S_{570}	0.4531	0.9041	0.4097	3
S_{582}	0.2531	1.0000	0.2531	4
S_{412}	0.8344	0.2753	0.2297	5
S_{546}	0.6281	0.2240	0.1407	6
S_{580}	0.2250	0.5923	0.1333	7
S_{454}	0.4781	0.2392	0.1144	8
S_{577}	0.2656	0.3755	0.0997	9
S_{593}	0.2031	0.4496	0.0913	10
S_{568}	0.4719	0.1315	0.0621	11
S_{445}	0.4813	0.1166	0.0561	12
S_{735}	0.2625	0.1487	0.0390	13
S_{476}	0.2844	0.0956	0.0272	14
S_{729}	0.2594	0.0976	0.0253	15
S_{404}	0.7000	0.0338	0.0237	16
S_{441}	0.5563	0.0411	0.0229	17
S_{600}	0.2375	0.0808	0.0192	18
S_{458}	0.4500	0.0421	0.0189	19
S_{469}	0.4656	0.0358	0.0167	20

Continued on the next page

Table 3.9: *(continued from the previous page)*

S_X	P_N	A_N	M_{PF}	P_R
S_{435}	0.5156	0.0192	0.0099	21
S_{581}	0.4750	0.0200	0.0095	22
S_{755}	0.2812	0.0232	0.0065	23
S_{753}	0.2375	0.0258	0.0061	24
S_{616}	0.3219	0.0164	0.0053	25
S_{503}	0.4500	0.0099	0.0045	26
S_{756}	0.2375	0.0099	0.0024	27
S_{428}	0.2594	0.0071	0.0018	28

Table 3.9 shows the results of the implementation Equations (2.1–2.3) to calculate the normalised priority (P_N), normalised activity (A_X), and maintenance factor (M_{PF}) respectively. Table 3.9 only contains the upper quartile subsystems of all the total subsystems of Table B.1 in Table 3.11. The results are visually presented in Figure 3.9 collected from Table 3.9 with the breakdown of user activity.

In Figure 3.9, subsystem S_{582} had the most user-generated events, with an average of 72.75 user-generated events per user. The lower P_N reduced its maintenance priority factor to the highest 4_{th} for all subsystems. The subsystem S_{538} had the 5^{th} highest number of recorded user-based activities and the most users with access to the subsystem. The higher P_N had a greater impact on its maintenance performance factor.

3.4.3 Case Study B results

Case Study B is implemented on an older system than Case Studies A and C and is primarily created in `PHP`. This system has the same type of user activities as Case Study A's Table 3.7. This system used multiple logging points to capture the types of users of Table 3.7.

The purpose of the use of numerous logging points was to:

- Ensure each subsystem's user activities could be captured since there was no central point in the software architecture to capture the request.
- The system comprised groups of smaller software systems where the logging points could be added.
- Some adjustments had to be made to the logging points to ensure that the quality of the log was maintained when it was captured.
- Consistency was also important, so each logging has several minor differences added to ensure it can consistently capture certain user-based events.

The logging points do not track any of the elements that the user interacted with in the system. Only a portion of the request parameters are tracked for user-based activities. The metadata will contain only the `RequestParameters` used in Listing 3.1. The breakdown of user activity types for this case study is shown in Figure 3.10.

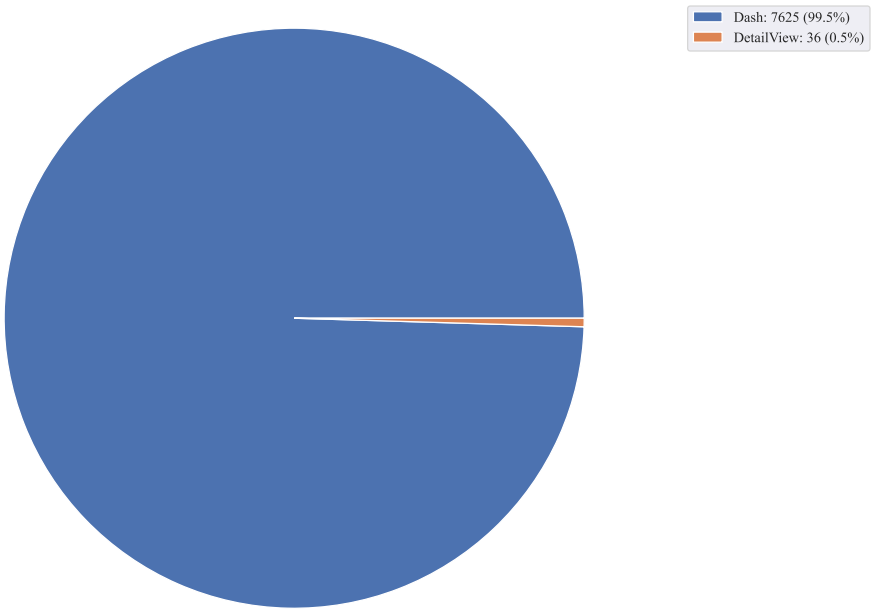

Figure 3.10: *User activity types breakdown of Case Study B*

In Figure 3.10, the `Dash` user activity type is approximately 99.5% of all user activities recorded. Case Study B has fewer subsystems that require the user to interact with the web page to accumulate data. The system mainly comprises web pages that display data only

for the user. There were no `Report` user activity types as the users did not create reports to export from these subsystems. Figure 3.11 breaks down each subsystem's utilisation user activity.

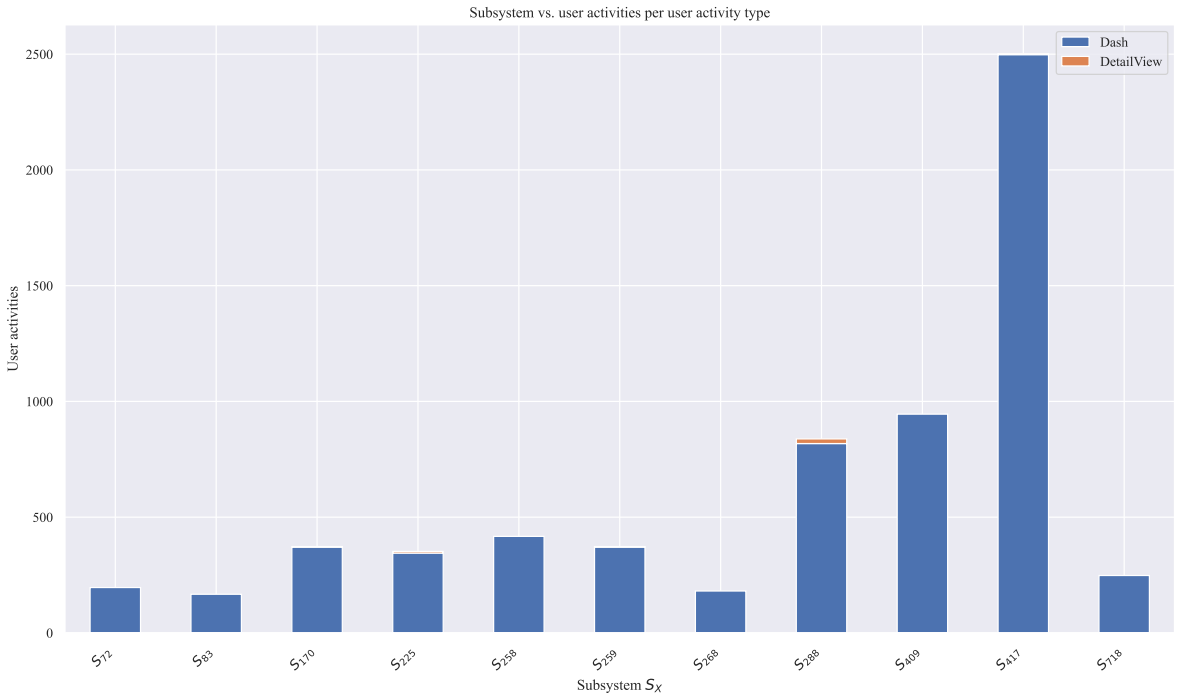

Figure 3.11: *System utilisation breakdown of Case Study B*

In Figure 3.11, all the subsystems have the majority `Dash` user activity type. Approximately 99.5% of the user activities are `Dash` user activity because the older system made in `PHP` is only used for monitoring energy systems. The subsystems have simple inputs, such as date pickers and category pickers. These input types only refresh the page, which causes more `Dash` activities to load new data for the user.

S_{417} has the most user activities, approximately 2.5 times higher than the second system, S_{409}, and has the most active users in this system. The log quality assessment for Case Study B is discussed in Table 3.10.

Table 3.10: *Logging quality assessment of Case Study B*

Req. ID	Description	Achieved	Comments
F/R 3.1.1	Log availability	Mostly	The log availability for Case Study B is: • Mostly *locally complete* since all the log attributes were available during the capturing phase of the event log. Each log point had its method of obtaining the same log attributes. • Mostly *globally complete* due to multiple logging mechanisms. They may have differences in their ability to identify a user-based event and capture the necessary log attributes to complete the log.
F/R 3.1.2	Log completeness	Mostly	The use of multiple logging points will yield different results in the same main software system. Some post-logging operations may be needed to fix and exclude some event logs.
F/R 3.1.3	Log extraction	✓	Although the availability and completeness of the event logs are generally achieved, the log extraction process is still possible.

Table 3.10 emphasises the issues of using multiple logging points with different operations to obtain user-based events. Case Study B used post-logging operations in the log analysis tool to exclude some unfixable logs, make applicable corrections, and fill in missing log attributes based on other log attributes.

Maintenance prioritisation

The maintenance prioritisation factor for the user activities in the upper quartile of the subsystem for Case Study B is calculated in Table 3.11.

Table 3.11: *Case Study B's upper quartile maintenance performance*

S_X	P_N	A_N	M_{PF}	P_R
S_{417}	1.0000	1.0000	1.0000	1
S_{288}	0.6756	0.3347	0.2261	2
S_{258}	0.9733	0.1663	0.1619	3
S_{259}	0.9733	0.1487	0.1448	4
S_{170}	0.7600	0.1487	0.1130	5
S_{225}	0.6089	0.1403	0.0855	6
S_{72}	0.6711	0.0780	0.0523	7
S_{83}	0.6711	0.0664	0.0445	8
S_{268}	0.5067	0.0720	0.0365	9
S_{718}	0.0089	0.0988	0.0009	10
S_{409}	0.0000	0.3774	0.0000	11

The upper quartile of the maintenance performance of the subsystems of Table B.2 in Table 3.11 is used to create the results of Table B.2 using Equations (2.1–2.3). A more significant disparity exists between the top A_N and the remaining subsystems' A_N. This significantly impacts the maintenance performance ranking, as S_{417} is ranked as the most critical system.

3.4.4 Case Study C results

Case Study C is a `APS.NET Core Web SDK` software system. The software system has seven primary user activity types described in Table 3.12.

Table 3.12: *Case Study A activity types*

Activity	Functional requirement	Description
`MenuAccessed`	F/R 1.2.1	Tracks user's navigation to a certain subsystem.
`LogoutAttempt`	F/R 1.2.2	Log-out attempt by the user to end their session using log-out controls.
`LogoutAttempt`	F/R 1.2.2	This user activity uses log-in page controls.
`ResetPassword`	F/R 1.2.2	This is a user activity when using any reset password page controls.
`SessionTracking`	F/R 1.2.2	Case Study C has some data stored in its session. Any changes to these data are tracked when the user interacts with any controls on the web page to change certain session data.
`CustomControls`	F/R 1.2.3	Case Study C has some custom HTML elements that have the same functionality as the `HTMLElement` user activity type.

Continued on the next page

Table 3.12: *(continued from the previous page)*

Activity	Functional requirement	Description
`HTMLElement`	F/R 1.2.3	This general activity type is for all additional activities that the user initiates that send *HTTP requests* back to the server. This type of user is primarily associated with HTML elements that the user uses to interact with the subsystem that is: • SpanClicked, • ButtonClicked, • DivClicked, • HyperLinkClicked, • ListClicked, • LabelClicked, • ImageClicked, • FormInput, • SelectClicked

For Case Study C, the user activities listed in Table 3.12 have been expanded to include the captured HTML elements with which the users interact. As Table 3.6 states, Case Study C is an administrative software system configuring and managing Systems A and B. Compared to the monitoring software subsystems of Case Study A and B, the user interaction in Case Study C is more complex.

To facilitate analysis for maintenance prioritisation, the increased types of user activity in Table 3.12 can be grouped. The `HTMLElement` type is a grouped user activity type of smaller individual user activity types. Despite their differences, these activity types share a standard base functionality that primarily shows the users' engagement with the system.

Case Study C uses a single logging point on the server side to capture the user event logs. The logging point for this implementation of the logging mechanism makes use of an action filter to:

- ensures that the logging point is executed before the rest of the request is serviced by the function that is called by the subsystem,
- globally defines all the controllers for the system,

- capture any additional user activity attributes and
- saves the completed log into the database before the logging process is finally terminated.

This logging point is similar to the one used for Case Study A but differs only in the critical logging attributes it needs to capture. Due to the tag name of the HTML element used to define some of the user activities of Table 3.12, the client side uses a similar logging point to capture the HTML element that the user interacted with.

The client-side logging point adds the last or clicked HTML element that the user used for the user-based event and saves it in a custom request header. Figure 3.12 breaks down the user activity of Case Study C's activity types in Table 3.12.

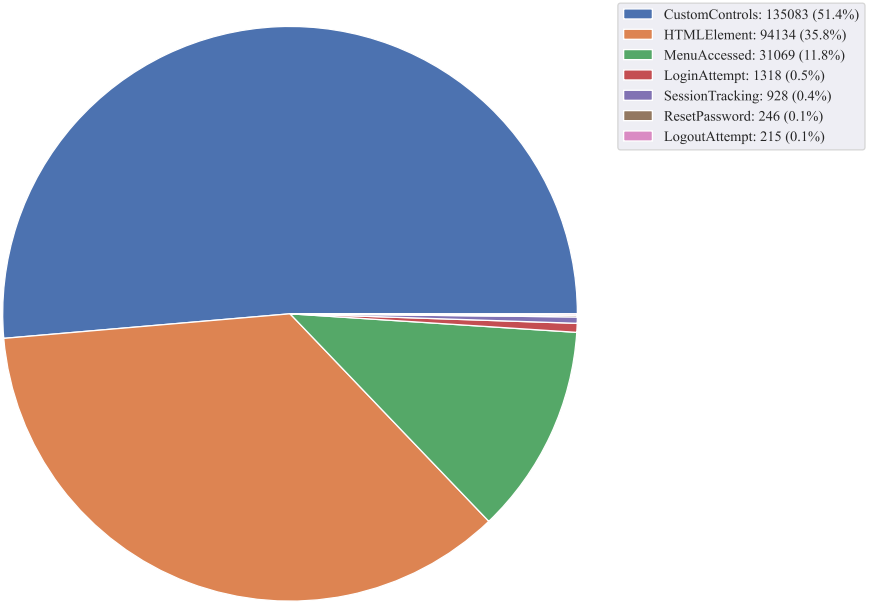

Figure 3.12: *User activity types breakdown of Case Study C*

According to the breakdown of user activities in Figure 3.12, most user activities for Case Study C is `CustomControl` user activity type. The more the user interacts with the system, the more the total amount of the general user activity types (`CustomControl` and `HTMLElement`) increases. Figure 3.13 is a breakdown of the user activity of each subsystem's utilisation.

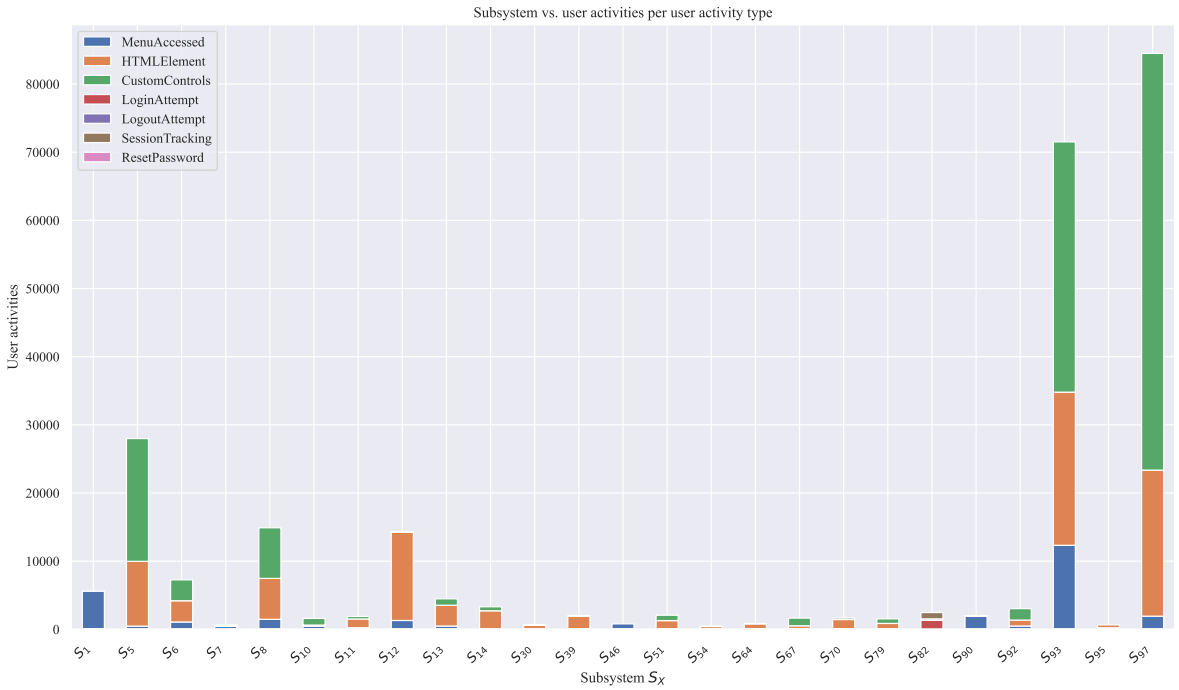

Figure 3.13: *System utilisation breakdown of Case Study C*

In Figure 3.13 the total captured user-based event logs are dominated by S_{97}, S_{93} and S_{12}. Most user-based events are general activity types for all the subsystems, as this case study is used for configuration purposes. Editing these configurations will create more general user activity types than other types of user activity. The log quality assessment for Case Study C is discussed in Table 3.13.

Table 3.13 indicated that logging attributes were missing for some types of user activity. This is not as severe as the incomplete logging attributes of Case Study B. It is expected that the main purpose of these activity types is to track log-in and log-out attempts.

Table 3.13: *Logging quality assessment of Case Study C*

Req. ID	Description	Achieved	Comments
F/R 3.1.1	Log availability	✓	The log availability for Case Study C is: • *Locally complete* due to most of the log attributes being available during the capturing phase of the event log. There are incomplete logs where log attributes were not obtained for user activity types, such as the `LoginAttempt`. The user's identity is not yet known on some of the captured event logs. As seen in Case Study A, this case study uses a central logging point for the client- and server-side. • *Globally complete* due to using a single log point on the client and server side of the main software system that captures events in most of the subsystems.
F/R 3.1.2	Log completeness	✓	The logging attributes are complete for all the subsystems, as no other post-logging operations are needed to fix some of the logs.
F/R 3.1.3	Log extraction	✓	Although the availability and completeness of the event logs are mostly achieved, the log extraction process is still possible.

Maintenance prioritisation

The maintenance prioritisation factor for the user activities in the upper quartile of the subsystem for Case Study C is calculated in Table 3.14.

Table 3.14: *Case Study C's upper quartile maintenance performance*

S_X	P_N	A_N	M_{PF}	P_R
S_{97}	1.0000	1.0000	1.0000	1
S_{93}	0.9931	0.8464	0.8405	2
S_{12}	0.5903	0.1694	0.1000	3
S_{8}	0.5625	0.1763	0.0992	4
S_{5}	0.2917	0.3312	0.0966	5
S_{6}	0.7083	0.0859	0.0608	6
S_{1}	0.8681	0.0660	0.0573	7
S_{82}	0.8681	0.0291	0.0253	8
S_{14}	0.5208	0.0391	0.0204	9
S_{90}	0.7778	0.0231	0.0180	10
S_{13}	0.3264	0.0529	0.0173	11
S_{92}	0.1736	0.0359	0.0062	12
S_{11}	0.2500	0.0223	0.0056	13
S_{10}	0.2778	0.0188	0.0052	14
S_{46}	0.4167	0.0095	0.0039	15
S_{67}	0.1181	0.0194	0.0023	16
S_{7}	0.2778	0.0070	0.0019	17
S_{70}	0.0833	0.0197	0.0016	18
S_{95}	0.1597	0.0076	0.0012	19
S_{39}	0.0417	0.0238	0.0010	20
S_{79}	0.0347	0.0182	0.0006	21
S_{64}	0.0625	0.0100	0.0006	22

Continued on the next page

Table 3.14: *(continued from the previous page)*

S_X	P_N	A_N	M_{PF}	P_R
S_{51}	0.0208	0.0245	0.0005	23
S_{54}	0.0486	0.0056	0.0003	24
S_{30}	0.0278	0.0077	0.0002	25

The upper quartile of the maintenance performance of the subsystems of Table B.3 in Table B.3 is used to create the results of Table B.3 using Equations (2.1–2.3). The results of Table 3.14 are visually presented in Figure 3.13.

The high usage of S_{97}, S_{93}, and S_{12} with a high number of users linked to each one increased their maintenance priority factor. Other systems such as S_6, S_1 and S_{82} have a higher priority normalisation than S_{12}, but they have a significantly lower normalisation of user activity.

3.4.5 Critical analysis results

Summary of comparison between case studies

The three case studies discussed respectively Sections 3.4.2 to 3.4.4 have similarities and differences. The logging mechanism used for Case Study A and Case Study C is similar, and both case studies make use of the MVC architecture discussed in Section 2.4.1:

- Both systems have a modified application of the Section 3.2.2 client-side event log to obtain the HTML element with which the user interacts.
- Using action filters in `C#` to create a single log point on the server side to capture user-based events.
- The log quality of these two case studies is similar since both systems would identify user-based events through the action filter used as a logging point.

The older subsystems of Case Study C use multiple logging points in the logging mechanism to capture logs. The same user-based event types defined in Table 3.7 of Case Study A were also identified. Case Study C has, therefore:

- More logging points than Case Studies A and B.
- A higher chance in the variance of the log quality than Case Study A and B's logging points. Different subsystems require modification of the logging points to ensure that the user-based event logs are captured for Case Study C.

- Decreased adaptability and maintainability due to the increased logging points.

Each case study had different breakdowns of their types of user activity. For Case Study A and B, which have the same kind of user activity, the analysis of Figures 3.8 and 3.10 had different results respectively. Most of Case Study B's event types were `Dash` due to how the system works, refreshing the entire page to obtain new data.

There weren't many `DetailView` activity types for these older subsystems with a more straightforward design as Case Study B's subsystem. Approximately 30% of Case Study B's activities were `DetailView`. The more complex subsystems requiring more user input increased this user type's share of the total captured logs.

These logs could have been categorised into other more descriptive user activity types, such as with Case Study C. This was not necessary for Case Study A because the purpose of its maintenance priority is more comparable to the software system in Case Study B, which has a similar operational goal.

All three case studies' user-based event logs are stored in a structured database with Case Study A and B using the same data structures. Case Study A and Case Study B use the same analysis procedures for the log analysis.

Case Study C's logs did not have a corresponding subsystem data table for log analysis. Additional post-logging operations group all the different request URLs into subsystems. For the more complex and more extensive software system used in Case Study C, it was preferable to use this method to categorise logs into subsystems using the request URLs target controller file.

Priority normalisation for Case Study A was superior and had a higher impact on the maintenance priority factor than Case Study A and C. Even if there were systems with a higher total user activity, the total number of active users had a more significant impact than the number of activities for most subsystems.

Case Study B's highest subsystem ranked for maintenance priority had all the highest normalised priority and user activities. This subsystem is a home page that all users can access and use to navigate the rest of the system.

Case Study C had two subsystems with a maintenance priority factor greater than 0.1. Even if many users have been active in a subsystem, the number of captured user activities impacted the maintenance priority factor more than in Case Studies A and B.

Value added per case study

In each case study, some obstacles must be overcome to create a suitable logging mechanism for log analysis. Some of the unique obstacles and the value added for each case study

include:

- **Case Study A**
  - This case study provides insightful data for management on which systems can be used for more strategic business decisions and the services provided to the end user.
- **Case Study B**
  - This case study made use of multiple logging points. Each logging point had unique obstacles to overcome to ensure that log quality was consistent and appropriate for log analysis.
  - There are multiple old subsystems for this case study. The lack of activity on some of these systems provides evidence that these systems are no longer used.
- **Case Study C**
  - The additional logging types for the user's session changes provided logs to monitor the user's access to the system.
  - The capture of the request parameters stored in the metadata log attributes provides additional information for developers to troubleshoot errors or bugs in the software system. Developers don't need direct aid from the user to understand what sequence of actions they perform to receive a certain response from the system.
  - Multiple subsystems provided the same functionality. The subsystems used more regularly are updated with any missing functionality from the less regularly used subsystems. This reduced the number of subsystems and possibly integrated similar functionality into one subsystem to improve the user experience.

Summary of positive points

From the observations made regarding each case study, some positive points can be summarised:

- Similar architecture software systems can use the same logging points that do not require too many modifications.
- Log quality for similar logging mechanisms is approximately the same for software system comparison.
- Log analysis tools can be used on all captured logs when implementing the correct log extraction process.
- Log analysis can be used for maintenance priority factor calculations.
- The lowest ranked maintenance priorities can be reviewed if they still offer value to the software system.

Summary of negative points

- Some systems needed additional logging points that were each modified to obtain the desired user-based event logs.
- Log quality may differ for different implementations of logging points. This can happen in the same software system with multiple logging points.
- Log quality may impact the maintenance priority factor since some variables can have extreme cases where values are notably high. Certain subsystems, such as a navigation page, do not have a meaningful impact on the user but are needed to use the rest of the system.

Functional requirements addressed

The functional requirements defined in Chapter 2 for Case Studies A, B, and C are addressed in Table 3.15.

Table 3.15: *Functional requirements addressed*

Req. ID	Case study		
	A	B	C
Log attributes (F/R 1)			
F/R 1.1.1	✓	✓	✓
F/R 1.1.2	✓	✓	✓
F/R 1.1.3	✓	✓	✓
F/R 1.1.4	✓	✓	✓
F/R 1.1.5	✓	✓	✓
F/R 1.1.6	✓	✓	✓
F/R 1.2.1	✓	✓	✓
F/R 1.2.2	✘	✘	✓
F/R 1.2.3	✓	✓	✓
F/R 1.3	✓	✓	✓
Logging point creation (F/R 2)			
F/R 2.1.1	✓	✓	✓
F/R 2.1.2	✓	Mostly	✓
F/R 2.1.3	✓	Mostly	✓
F/R 2.1.4	✓	✓	✓
F/R 2.2	✓	✓	✓

Continued on the next page

Table 3.15: *(continued from the previous page)*

Req. ID	Case study		
	A	B	C
Log analysis tools (F/R 3)			
F/R 3.1.1	✓	Mostly	✓
F/R 3.1.2	✓	Mostly	✓
F/R 3.1.3	✓	✓	Mostly
F/R 3.2.1	✓	✓	✓
F/R 3.2.2	✓	✓	✓
F/R 3.2.3	✓	✓	✓
Maintenance prioritising (F/R 4)			
F/R 4.1.1	✓	✓	✓
F/R 4.1.2	✓	✓	✓
F/R 4.1.3	✓	✓	✓
F/R 4.2.1	✓	✓	✓
F/R 4.2.2	✓	✓	✓
F/R 4.2.3	✓	✓	✓
F/R 4.2.4	✓	✓	✓

Table 3.15 indicates that the functional requirements for the logging attributes (F/R 1) defined in Table 2.2 for all three case studies were met. In each case study, only user-based events were captured for the user activity types defined for each case study, and the log attributes captured from the user-based event were obtained.

Case Study A and B did not meet the F/R 1.2.2 requirement. Session changes in user activity type were unnecessary for these case studies since internal and external users' actual interactions with the systems were more important, and no other session-related activities could contribute to the log analysis.

The functional requirement of the logging point (F/R 2) for Case Study A and C addressed all sub-functional requirements. Case Study B did not meet all requirements as F/R 2.1.2 and F/R 2.1.3 are not fully met. In this older system, using multiple logging points requires modifications for groups or individual subsystems. This can cause inconsistencies in log quality, as some potential user-based event logs may not be consistently identified.

Log analysis functional requirements (F/R 3) is done using and creating custom log analysis tools. Since user-based events could be efficiently obtained from the software system, the quality of the log (F/R 3.1) for Case Study A and B was complete.

Case Study B had lower log quality, as the availability and completeness of the logs were lower due to the use of multiple log points. The log extraction (F/R 3.1.3) for Case Study C was achieved mainly because additional post-logging activities with the log analysis tool had to be used to create subsystems using the logs.

Through the log analysis, maintenance priority (F/R 4) could be performed for each case study. The results of each case study were evaluated by comparing the software maintenance prioritisation with the other case studies.

Gaps identified

By analysing the results of the case studies and the test, these gaps were identified for this study:

1. Log quality is important for further analysis of user-based events. Improvement in the implementation of the logging mechanism can increase the following:

   - Accuracy of the logging mechanism to capture logs,
   - trustworthiness of more complete consistent logs, and
   - decreased performance impact on the rest of the software system.

   Applying more fundamentals of Figure 1.5 could have improved the logging mechanism. This should also improve the log quality, especially for implementing Case Study B's logging mechanism.

2. The log analysis used Equations (2.1–2.3) to rank the maintenance priority of each subsystem. Each case study had outliers for one of the two main parameters used for Equation (2.1). Some gaps for this study with these parameters include:

   - The normalised priority (P_N) uses the total active users. Some users were internal and external clients, and others were software developers. Adding weight to some of these user types can improve the impact that P_N has on M_{PF}. Some users, such as developers, are significant users who can't determine whether a subsystem is important when external client users pay for the software system to use it.

- Normalised user activities (A_N) have a similar problem to P_N. There are less important user activity types, and adding weight to how important each user activity type is can improve log analysis. For Case Study C, some of the similar user activity types were grouped to form one user activity type. This can be used more, or user activity types should be better defined to be more distinct.

3.5 Conclusion

This chapter explored the implementation of the methodology defined in Chapter 2 to create a logging mechanism for different case studies. A test system was used to initially verify the development of the solution before it was implemented in three different case studies.

Log analysis was performed for all logs obtained for October 2022. The following was carried out for each case study:

- Identification of the type of activity of the user,
- implementations of log-points,
- log analysis, comprising:
  - user activity type breakdown
  - priority normalisation
  - normalisation of user activity
  - maintenance priority factor calculation, and
- maintenance priority ranking and recommendations.

All results were compared for the case study. The overall result of the log analysis is that implementing a user-based event-logging mechanism can aid in prioritising software maintenance. Particular gaps that can improve maintenance prioritising were identified.

Chapter 4

Conclusion

4.1 Discussion

4.1.1 Contributions made to literature

Chapter 1 highlighted the importance of software maintenance throughout the life cycle of most software systems. A gap in the literature was identified in Section 1.2.5 from the studies obtained on software maintenance.

Using event logging to create a log analysis for utilising the software systems is a possible solution to make software maintenance prioritisations. A state of the art analysis was performed for the literature on software maintenance, event logging, and log analysis and outlined in Table 1.10. This study will be referred to as *Study U*, filling in the gaps in the literature obtained with the method in Chapter 2 that is included in the updated state of the art in Table 4.1.

Table 4.1 highlights a divide between software maintenance, event logging, and log analysis in the state of the art. *Study U* can contribute to the literature by providing a comprehensive method that includes all three main research topics.

Event logging and log analysis as a primary topic overlap in the literature more often than software maintenance. Research on software maintenance was identified as being limited by a lack of third-party validation, not always trying to improve or build existing software maintenance models, little comparison with other literature, and difficulty replicating due to private data sets or custom tools.

The obtained studies that discussed software maintenance frequently identified that the efficiency of implementing software maintenance is essential. Some studies attempted to use specific software maintenance models on certain software maintenance types for better efficiency.

Various studies tried to create a task distribution for developers to improve software maintenance efficiency. Other studies attempted to isolate certain software maintenance resource costs in the entire Software Development Life-Cycle (SDLC).

Additionally, these studies did not attempt to provide the method needed to prioritise these software systems. The method requires a logging mechanism to do the log analysis, which is not present in software maintenance-related studies.

Study U does not explicitly use software maintenance models as it is irrelevant to prioritising software systems. For this reason, the model's state of the art topic is marked as partially achieved. From the software maintenance problems identified in the literature, the requirements for the log analysis are created for the software maintenance prioritisation.

The methods for the event logging mechanisms are not fully described or use third-party

Table 4.1: *State of the art*

Ref.	Software maintenance			Event logging		Log analysis
	Models	Problems	Prioritisation	Parsing	Points	Utilisation
[9]	Partial	✓	✓	✘	✘	✘
[10]	Partial	Partial	✓	✘	✘	✘
[12]	Partial	✓	✓	✘	✘	✘
[16]	✘	✓	✓	✘	✘	✘
[18]	✓	Partial	✘	✘	✘	✘
[24]	Partial	✓	✓	✘	✘	✘
[28]	Partial	✘	✓	✘	✘	✘
[32]	✘	✘	✘	✓	Partial	✘
[56]	✘	✘	✘	✓	✓	✓
[35]	✘	✘	✘	Partial	Partial	✘
[42]	✘	✘	✘	✓	✓	✘
[43]	✘	✘	✘	Partial	Partial	✘
[51]	✘	✘	✘	✓	✓	Partial
[50]	✘	✘	✘	✓	✓	Partial
[54]	✘	✘	✘	✘	✘	✓
[57]	✘	✘	✘	✘	✘	✓
*U**	Partial	✓	✓	✓	✓	✓

* *Study U: Using user-based activity logging and analysis to prioritise software maintenance*

tools. Software maintenance does not overlap with this topic, as most event logs are for system diagnostics.

There were logging mechanisms that captured user-based logs used for user-behavioural analysis instead of system utilisation analysis. *Study U* contributes to the literature on system utilisation analysis for software maintenance when prioritised.

4.1.2 Value added to industry

The method in Chapter 2 is used in three different case studies in Section 3.4. Each of these case studies explored the various applications of the generic logging mechanism to obtain user-based events. The results proved that the logging mechanism could get the desired user-based activities. However, additional adaptations were needed for each case study to

ensure that log quality was acceptable for consistent and reliable log analysis.

These adaptations were due to the software environment (software languages and design methodologies used) and the purpose of the software system. Older systems require different logging points to produce the same result as newer software systems that use less or one logging point.

These systems used the same user-based event type with similar operational software use cases. Systems with the same software architecture but different functional use cases also had various adaptations to their logging mechanism. These adaptations ensured that the desired log attributes were correctly obtained for the log analysis.

The maintenance prioritisation recommendations for each case study used the defined generic methodology. The log analysis for each case study was similar to the log requirements for the user-based utilisation for all the case studies and were only different for each of their user-based event types. These user-based event types represent the operational use cases for the case study to observe the kind of utilisation of the subsystems further.

The results proved that the generic methodology defined in Chapter 2 could be implemented on different software systems with alternative operational use cases for each case study. In the web development industry, software systems similar to the case studies and different software systems will benefit from implementing this methodology to improve software maintenance decisions using these recommendations for prioritisation.

4.1.3 Validation strategy

The following five-step validation strategy is used for the outcomes of this study:

I. Revisit the literature gap defined in Section 1.2.5.

II. Revisit the original problem statement defined in Section 1.3.

III. Revisit the study objectives defined in Section 1.4.

IV. Reflect on the results of the study methodology in Chapter 3.

V. Determine whether a solution to the original problem statement has been presented.

This study is validated by:

I. Gap in literature

The literature gap defined in Section 1.2.5 for implementing software maintenance is summarised as:

> *Implementing software maintenance efficiently with the limited available resources organisations or individuals have is crucial to improving the efficiency of software maintenance efforts. Assisting developers in making better-informed decisions about which systems to prioritise maintenance can improve software maintenance.*

II. Original problem statement

The original problem statement defined in Section 1.3 to make software maintenance prioritisations:

> *Software maintenance is a problem in the industry due to how inefficiently developers prioritise maintenance activities. A proven method to monitor software behaviours is event logging. The logging mechanism and log analysis must be explicitly designed for user-based events to improve software maintenance.*

III. Study objectives

The study objectives for this study that were defined in Section 1.4 and have been met are outlined Table 4.2.

Table 4.2: *Study validation*

Objective ID	Objective	Section	Objective met
	Literature Objectives		
L1	Investigate what will define a user-based event log and how to obtain the needed log attributes for a comprehensive user utilisation log analysis.	Sections 2.3.1 and 2.4.1	✓
L2	Investigate where to place logging points in a software system to capture user-based event logs.	Sections 2.3.2 and 2.4.1	✓
L3	Investigate how to evaluate the log quality of the stored user-based event logs.	Section 2.3.3	✓

Continued on the next page

Table 4.2: *(continued from the previous page)*

Objective ID	Objective	Section	Objective met
L4	Investigate how to implement maintenance prioritisation.	Section 2.3.4	✓
	Empirical Objectives		
E1	Implement the method to create a user-based logging mechanism on a test system: (a) Define the user activity types and create the needed log attributes for a user-based event log. (b) Implement the logging points at strategic locations to capture the relevant log attributes to create event logs. (c) Use or create a log analysis tool to evaluate the log quality of the captured logs. (d) Implement a software maintenance prioritisation method in the log analysis.	Section 3.2	✓
E2	Verify the results for the test system's software maintenance prioritisation.	Section 3.3	✓
E3	Validate the software maintenance prioritisation method by critically analysing the case studies' results.	Section 3.4	✓

In Table 4.2, for objective L1, the characteristics of the event log are defined with the expected log attributes needed for the log analysis. The characteristics of the event log focus only on obtaining user-based event logs with the necessary qualities that the logging point can capture and store.

For objective L2, the log attributes of each user-based event must be obtained from the

software system using a logging point. The logging points are placed strategically in the software system where they have little impact on the performance of the software system. The requirements for the event logs are created to guide developers in making the logging points to capture the event logs.

For objective L3, the quality of the logs is essential for log analysis. This objective ensures that the captured event logs are complete, accurate, and available when the logs are extracted by the log analysis tool. The method requirements were defined to adhere to the log quality specifications.

For objective L4, maintenance prioritisation is determined. The calculations needed for this log analysis are explained for this objective that needs to be executed in the log analysis. Prioritisation of each subsystem will be calculated in the specified time frame. For this objective, the requirements for which logs to use are also defined, as some types of user activity can be excluded based on the software system used.

Empirical objectives use the method in a test system to verify the results and implement the method in the case studies. It also reflects on the study methodology using the results to determine whether this solution has resolved the problem statement.

For objective E1, the following sub-objectives are met for the implementation of the method on the test system:

(a) The user activity types and log attributes for the test system were defined. A logging point(s) was placed on the client or server side for the test system.

(b) This strategic placement of the logging point(s) was determined by what log attributes need to be obtained in certain stages of the software system's logging mechanism, structure, and complexity. The logging points store the event logs in a structured database.

(c) A log analysis tool is either created or a suitable third-party tool is used to analyse the extracted logs. Log quality is also checked in this objective to determine whether the logs meet the standard set by the requirements in L3.

(d) Maintenance prioritisation is calculated for each subsystem for this sub-objective using L4.

For objective E2, the results of implementing the method in the test system are verified in this objective. Objectives L1, L2, L3, and L4 verified if the technique could create software maintenance prioritisations for the test system by comparing it to the expected results.

For objective E3, the verified method of E1 is applied to multiple case studies and the results are evaluated. A critical analysis of the case studies is performed to validate

the software maintenance prioritisation for this study, utilising the case studies' results. With this validation strategy, it is validated that this study meets the study objectives with the created solution for the original problem statement.

IV. Reflection on methodology

In Chapter 2, the development of the functional requirements of the solution is made in Section 2.2. This provides a generic method to create a logging mechanism and analyse the obtained user-based logs. A test logging mechanism is designed with a log analysis of usage to validate the development of functional requirements for the solution in Table 2.1.

This logging mechanism aims to implement each sub-functional requirement in a test system by testing specific inputs to verify expected outputs. The following validation method was used for this test implementation in Section 3.2:

- Identifying and creating the log attributes needed for the log analysis (F/R 1). The software system's key log attributes are precisely defined for maintenance prioritisation. The user-activity type forms the base requirement for a user-based log.
- Logging points are made to capture these logging attributes at specific locations in the software system (F/R 2). The placement of the logging points to capture user-based logs is verified if it can be done consistently, accurately, and discretely without impacting the software system's performance.
- Log analysis is verified using a third-party log analysis tool or the implementation of a log analysis (F/R 3). For the log analysis to be successful, the quality must be acceptable. This is verified by the logging points' performance and the usability of the logs without too many post-logging corrections made.
- Creating software maintenance prioritising (F/R 4) from the results of log analysis. In the log analysis, the different subsystems' maintenance priority (M_{PF}) are calculated from the normalised total active users (P_N) multiplied by the normalised total user activity (A_N) for a specified subsystem in Equations (2.1–2.3).

  These results are verified with the test case study and Case Studies A, B, and C in Section 3.4 on different software systems with different operational use cases. The results obtained for the maintenance priority validate the previous implementation using the defined user-based logs to perform the log analysis for the maintenance priority.

V. Conclusion

The following outcomes on how the solution has been presented to solve the original problem statement for this study are presented:

- The solution in this study aims to prioritise software maintenance by using event logging and log analysis.
- The study results showed that the solution could create a logging mechanism for prioritisation of software maintenance, as stipulated in the study objectives of Table 4.2.
- Therefore, the need to develop a method for log analysis for software maintenance by creating a suitable logging mechanism to capture user-based event logs has been addressed by the study objectives.
- The original problem statement defined in Section 1.3 has been successfully addressed by the study objectives.
- Finally, the identified gap has been addressed and fulfilled by providing a solution to the original problem statement.

4.2 Recommendations

In Section 3.4, the case studies highlighted the method's limitations.

4.2.1 Automated logging and AI tools

In Section 1.2.4, automated logging using AI (artificial intelligence) was stated as not being part of the scope of this research due to the reasons listed in the section. However, using AI tools could potentially address the research problem and enhance the creation of a suitable log mechanism and analysis.

Future research should delve into the development of user activity logging solutions that leverage AI tools for automated logging. AI tools can further integrate machine learning techniques to automatically analyse and derive meaningful patterns from large-scale user activity logs. This will provide valuable insights and predictions for the maintenance priority.

4.2.2 Logging quality

Table 1.3 describes the event log quality as an essential component of the logging mechanism. Event logs must be accurate, manage complex structures, and be consistent and complete.

For the methodology used, not all dimensions of Figure 1.5 are used to design the logging mechanism.

This presents the functional requirement of Case Study B's logging points (F/R 2). The multiple logging points introduced inconsistencies in the event logs for the groups of subsystems used. There are studies on improving event log quality, but creating an event log quality model specifically for user-based event logging can increase log quality.

Some of the defined event log quality model requirements specified in Figure 1.5 still need to be fully integrated into the method. Some of these requirements can add value to log quality.

4.2.3 Maintenance prioritisation

The maintenance prioritisation (F/R 4) can be improved upon in multiple regards. Equations (2.1–2.3) use the full user-based event logs per subsystem and the total users linked per system as their base variables.

From the results observed in the case studies in Section 3.4, the base variables had an overwhelming impact on the prioritisation factor. Introducing additional variables that represent alternative factors from the log attributes can improve the accuracy of maintenance prioritisation. An important variable that could have been used is the type of user activity of each case study.

User activity types

Throughout the study, the type of user activity was essential to describe what can be classified as a user event. This primary log attribute can add another dimension to the results, as some user activities may be more important than others.

For software systems, as in Case Study C, where most types of user activity were grouped, the results can be beneficial. There are some differences between some activity types.

User and activity parameters

Normalisation is used in Equations (2.1–2.3) to make a comparable scale for both main variables for software maintenance prioritisation. While normalisation yielded similar results, there were numerous outliers on each variable's low and high ends. The data do not follow a specific distribution or pattern as S_{582} of Case Study A with a normalised activity of 1. This subsystem had the highest normalised activity but had the 4^{th} highest maintenance priority due to its lower unique active user base.

Using different techniques to formulate a software maintenance priority factor may have placed seen in S_{582} higher if the effect of total unique users was not used as the primary

prioritisation factor for this situation.

Use of other classification strategies

Normalisation was used for maintenance priority calculations. Using other statistical methods can yield improved prioritisation. Only one prioritisation method was used. A need exists to enhance prioritisation by comparing different prioritisation methods.

4.3 Conclusion

4.3.1 In summary

- There is a need to improve software maintenance activities in the industry, but software maintenance prioritisation is still a problem for most software developers.
- Event-based logging is a proven method for gathering valuable information about a software system.
- A log analysis of the utilisation of the software system can provide the evidence required to prioritise software systems based on the extracted log data.
- Creating a user-based event logging mechanism to implement a system utilisation log analysis solves the identified problem.
- Three different case studies with two further operational use cases verified the methodology designed for this study.

4.3.2 Conclusion

Analysing user-based event logs can improve software maintenance resource management by prioritising maintenance tasks through a comprehensive log analysis.

References

[1] C. Gralha, D. Damian, A. I. Wasserman, M. Goulão, and J. Araújo, "The evolution of requirements practices in software startups," *Proceedings - International Conference on Software Engineering*, pp. 823–833, 2018.

[2] P. M. Khan and M. M. Beg, "Extended decision support matrix for selection of sdlc-models on traditional and agile software development projects," *International Conference on Advanced Computing and Communication Technologies, ACCT*, pp. 8–15, 2013.

[3] N. Al-Saiyd and E. Zriqat, "Analyzing the Impact of Requirement Changing on Software Design," *European Journal of Scientific Research*, vol. 136, no. February, 2015.

[4] C. Ackermann, M. Lindvall, and G. Dennis, "Redesign for flexibility and maintainability: a case study," *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 259–262, mar 2009.

[5] D. Reimanis and C. Izurieta, "Towards Assessing the Technical Debt of Undesired Software Behaviors in Design Patterns," *Proceedings - 2016 IEEE 8th International Workshop on Managing Technical Debt, MTD 2016*, pp. 24–27, 2016.

[6] M. De Leon-Sigg, S. Vazquez-Reyes, and D. Rodriguez-Avila, "Towards the use of a framework to make technical debt visible," *Proceedings - 2020 8th Edition of the International Conference in Software Engineering Research and Innovation, CONISOFT 2020*, pp. 86–92, 2020.

[7] W. Snipes, S. L. Karlekar, and R. Mo, "A case study of the effects of architecture debt on software evolution effort," *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, pp. 400–403, 2018.

[8] M. Wiese, M. Riebisch, and J. Schwarze, "Preventing Technical Debt by Technical Debt Aware Project Management," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, may 2021, pp. 84–93. [Online]. Available: http://arxiv.org/abs/2103.10317https://ieeexplore.ieee.org/document/9462991/

[9] E. E. Ogheneovo, "On the Relationship between Software Complexity and Maintenance Costs," *Journal of Computer and Communications*, vol. 02, no. 14, pp. 1–16, 2014.

[10] L. Tang, Y. G. Mei, and J. J. Ding, "Metric-based tracking management in software maintenance," *2nd International Workshop on Education Technology and Computer Science, ETCS 2010*, vol. 1, pp. 675–678, 2010.

[11] T. F. Thamburaj and A. Aloysius, "Models for Maintenance Effort Prediction with Object-Oriented Cognitive Complexity Metrics," in *2017 World Congress on Computing and Communication Technologies (WCCCT)*. IEEE, feb 2017, pp. 191–194.

[12] H. M. Sneed, "A cost model for software maintenance & evolution," *IEEE International Conference on Software Maintenance, ICSM*, pp. 264–273, 2004.

[13] D. Port and B. Taber, "Actionable Analytics for Strategic Maintenance of Critical Software: An Industry Experience Report," *IEEE Software*, vol. 35, no. 1, pp. 58–63, 2017.

[14] S. Mamone, "The IEEE standard for software maintenance," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 1, pp. 75–76, jan 1994.

[15] R. Hasan, S. Chakraborty, and J. Dehlinger, "Examining software maintenance processes in small organizations: Findings from a case study," *Studies in Computational Intelligence*, vol. 377, pp. 129–143, 2012.

[16] Z. Stojanov, J. Stojanov, D. Dobrilovic, and N. Petrov, "Trends in software maintenance tasks distribution among programmers: A study in a micro software company," in *2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY)*. IEEE, sep 2017, pp. 000 023–000 028.

[17] N. Niu, S. Brinkkemper, X. Franch, J. Partanen, and J. Savolainen, "Requirements engineering and continuous deployment," *IEEE Software*, vol. 35, no. 2, pp. 86–90, 2018.

[18] M. Galster, C. Treude, and K. Blincoe, "Supporting Software Architecture Maintenance by Providing Task-Specific Recommendations," *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pp. 370–372, 2019.

[19] L. Ping, "A quantitative approach to software maintainability prediction," *Proceedings - 2010 International Forum on Information Technology and Applications, IFITA 2010*, vol. 1, pp. 105–108, 2010.

[20] U. Kumar, D. Galar, A. Parida, C. Stenström, and L. Berges, "Maintenance performance metrics: a state-of-the-art review," *Journal of Quality in Maintenance Engineering*, vol. 19, no. 3, pp. 233–277, aug 2013.

[21] C. Rahmani and D. Khazanchi, "A study on defect density of open source software," *Proceedings - 9th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2010*, pp. 679–683, 2010.

[22] S. M. A. Shah, M. Morisio, and M. Torchiano, "An overview of software defect density: A scoping study," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 1, pp. 406–415, 2012.

[23] M. Alenezi and M. Zarour, "Does Software Structures Quality Improve over Software Evolution ? Evidences from Open - Source Projects," *Special issue on "Computing Applications and Data Mining" International Journal of Computer Science and Information Security (IJCSIS*, vol. 14, no. 1, pp. 61–75, 2016.

[24] V. Lenarduzzi, A. Sillitti, and D. Taibi, "Analyzing Forty years of software maintenance models," *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pp. 146–148, 2017.

[25] D. Garlan, "Software Architecture: a Roadmap David Garlan," *Design*, 1999.

[26] L. R. Vijayasarathy and C. W. Butler, "Choice of Software Development Methodologies: Do Organizational, Project, and Team Characteristics Matter?" *IEEE Software*, vol. 33, no. 5, pp. 86–94, 2016.

[27] Y. Ren, X. Tao, Z. Liu, and X. Chen, "Software maintenance process model and contrastive analysis," *Proceedings - 2011 4th International Conference on Information Management, Innovation Management and Industrial Engineering, ICIII 2011*, vol. 3, pp. 169–172, 2011.

[28] J. Araujo, C. Melo, F. Oliveira, P. Pereira, and R. Matos, "A Software Maintenance Methodology: An Approach Applied to Software Aging," *15th Annual IEEE International Systems Conference, SysCon 2021 - Proceedings*, 2021.

[29] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 806–821, 2013.

[30] G. Rong, S. Gu, H. Zhang, D. Shao, and W. Liu, "How is logging practice implemented in open source software projects? A preliminary exploration," *Proceedings - 25th Australasian Software Engineering Conference, ASWEC 2018*, pp. 171–180, 2018.

[31] S. Levin and A. Yehudai, "Visually exploring software maintenance activities," *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, pp. 110–114, 2019.

[32] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, pp. 121–130, 2019.

[33] F. Baccanico, G. Carrozza, M. Cinque, D. Cotroneo, A. Pecchia, and A. Savignano, "Event Logging in an Industrial Development Process: Practices and Reengineering Challenges," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, no. i. IEEE, nov 2014, pp. 10–13.

[34] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry Practices and Event Logging: Assessment of a Critical Software Development Process," *Proceedings - International Conference on Software Engineering*, vol. 2, pp. 169–178, 2015.

[35] G. Rong, Q. Zhang, X. Liu, and S. Gu, "A Systematic Review of Logging Practice in Software Engineering," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2017-Decem, pp. 534–539, 2018.

[36] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne, "CRUDE: Combining Resource Usage Data and Error Logs for Accurate Error Detection in Large-Scale Distributed Systems," *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 51–60, 2016.

[37] J. Dwyer and T. M. Truta, "Finding anomalies in windows event logs using standard deviation," *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, COLLABORATECOM 2013*, pp. 563–570, 2013.

[38] D. Evangelin Geetha, T. V. Suresh Kumar, and K. Rajani Kanth, "Predicting performance of software systems during feasibility study of software project management," *2007 6th International Conference on Information, Communications and Signal Processing, ICICS*, pp. 1–5, 2007.

[39] W. Song, X. Xia, H.-A. Jacobsen, P. Zhang, and H. Hu, "Efficient Alignment Between Event Logs and Process Models," *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 136–149, jan 2017.

[40] A. C. Pathan and M. A. Potey, "Detection of malicious transaction in database using log mining approach," *Proceedings - International Conference on Electronic Systems, Signal Processing, and Computing Technologies, ICESC 2014*, pp. 262–265, 2014.

[41] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering (Software Engineering Group, Department of Computer Science, Keele . . . ," Keele University, Tech. Rep. January, 2007.

[42] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 415–425, 2015.

[43] M. O. Kherbouche, N. Laga, and P. A. Masse, "Towards a better assessment of event logs quality," *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, 2017.

[44] S. Al-Fedaghi and F. Mahdi, "Events Classification in Log Audit," *International journal of Network Security & Its Applications*, vol. 2, no. 2, pp. 58–73, 2010.

[45] M. J. Jans, M. G. Alles, and M. A. Vasarhelyi, "Process Mining of Event Logs in Auditing: Opportunities and Challenges," *SSRN Electronic Journal*, no. August 2020, 2012.

[46] W. Van Der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[47] Y. A. Bekeneva, "Algorithm for Generating Event Logs Based on Data from Heterogeneous Sources," in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. IEEE, jan 2020, pp. 233–236.

[48] G. Rong, Y. Xu, S. Gu, H. Zhang, and D. Shao, "Can You Capture Information As You Intend To? A Case Study on Logging Practice in Industry," in *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*. Institute of Electrical and Electronics Engineers Inc., sep 2020, pp. 12–22.

[49] T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang, and Y. Liu, "Machine Deserves Better Logging: A Log Enhancement Approach for Automatic Fault Diagnosis," *Proceedings - 29th IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2018*, pp. 106–111, 2018.

[50] K. Slaninová, "User behavioural patterns and reduced user profiles extracted from log files," *International Conference on Intelligent Systems Design and Applications, ISDA*, pp. 289–294, 2014.

[51] A. Hasiloglu and A. Bali, "Central audit logging mechanism in personal data web services," *6th International Symposium on Digital Forensic and Security, ISDFS 2018 - Proceeding*, vol. 2018-Janua, pp. 1–3, 2018.

[52] P. Dhanalakshmi, K. Ramani, and B. E. Reddy, "The Research of Preprocessing and Pattern Discovery Techniques on Web Log Files," *Proceedings - 6th International Advanced Computing Conference, IACC 2016*, pp. 139–145, 2016.

[53] G. Kocsis and P. Ekler, "Analyzing the resource requirements of usage statistics gathering on online newspapers," *CINTI 2012 - 13th IEEE International Symposium on Computational Intelligence and Informatics, Proceedings*, pp. 213–218, 2012.

[54] M. Waqar and D. Rafiei, "Tracking User Activities and Marketplace Dynamics in Classified Ads," *Proceedings - 2016 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2016*, pp. 522–525, 2017.

[55] G. Paliouras, C. Papatheodorou, V. Karkaletsis, C. Spyropoulos, and P. Tzitziras, "From Web usage statistics to Web usage analysis," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, pp. 159–164, 1999.

[56] J. Bogatinovski and O. Kao, "Auto-Logging: AI-centred Logging Instrumentation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, may 2023, pp. 95–100. [Online]. Available: https://ieeexplore.ieee.org/document/10173904/

[57] M. Kumar and Meenu, "Analysis of visitor's behavior from web log using web log expert tool," in *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*. IEEE, apr 2017, pp. 296–301.

[58] P. R. Anish, B. Balasubramaniam, J. Cleland-Huang, R. Wieringa, M. Daneva, and S. Ghaisas, "Identifying Architecturally Significant Functional Requirements," *Proceedings - 5th International Workshop on the Twin Peaks of Requirements and Architecture, TwinPeaks 2015*, pp. 3–8, 2015.

[59] C. T. K.-H. Stadtländer, "Qualitative, Quantitative, and Mixed-Methods Research," *Microbe Magazine*, vol. 4, no. 11, pp. 485–485, 2009.

[60] M. Jailia, A. Kumar, M. Agarwal, and I. Sinha, "Behavior of MVC (Model View Controller) based Web Application developed in PHP and .NET framework," in *2016 International Conference on ICT in Business Industry & Government (ICTBIG)*. IEEE, 2016, pp. 1–5.

[61] M. X. Gu and K. Tang, "Comparative analysis of WebForms MVC and MVP architecture," *2010 2nd Conference on Environmental Science and Information Application Technology, ESIAT 2010*, vol. 2, pp. 391–394, 2010.

[62] S. K. Patro and K. K. Sahu, "Normalization: A Preprocessing Stage," *Iarjset*, pp. 20–22, 2015.

Appendix A

Logging practise in software engineering

Providing a guide for software engineers and developers to implement a suitable logging mechanism in their software systems has proven to be a vital tool for both industrial use and academic progress [30]. [30] conducted a study to review published articles on logging practises and improve the performance and efficiency of logging implementation. From their research, they established selection criteria to include (as in Table A.1) and exclude (as in Table A.2) academic papers on logging practises [30, 35].

Rong's selection criteria yielded numerous research articles on logging practises applied in the industry, either through creating a new logging mechanism or optimising existing logging mechanisms. Reviewing 41 identified articles, they found that many practitioners and researchers recognise the importance of logging practises in software engineering. However, a lack of guidance is available to provide software engineers or developers with the tools necessary to create or improve efficient logging mechanisms [30, 42].

Table A.1: *G. Rong's inclusion selection criteria [30]*

Identification	Criteria
I1.	Publications that investigate the methodology for logging practice.
I2.	Publications that investigate the tools, frameworks, and systems which support logging practice.
I3.	Publications that propose a standard for logging practice.
I4.	Publications that are peer-reviewed (conference paper, journal article).
I5.	Publications that are primary studies on logging practice.

Table A.2: *G. Rong's exclusion selection criteria [30]*

Identification	Criteria
E1.	Publications that investigate log analysis.
E2.	Publications that investigate the usage of logs.
E3.	Publications that investigate the technologies for logging user behaviours.
E4.	Publications that are not written in English.
E5.	Additionally, short papers, demos or industry publications are excluded.

Figure A.1 shows the distribution of the 41 published papers obtained for [30] research on logging practises. Event logging plays an increasingly important role in modern software systems; research on logging practises in software engineering increased between 1990 and 2017.

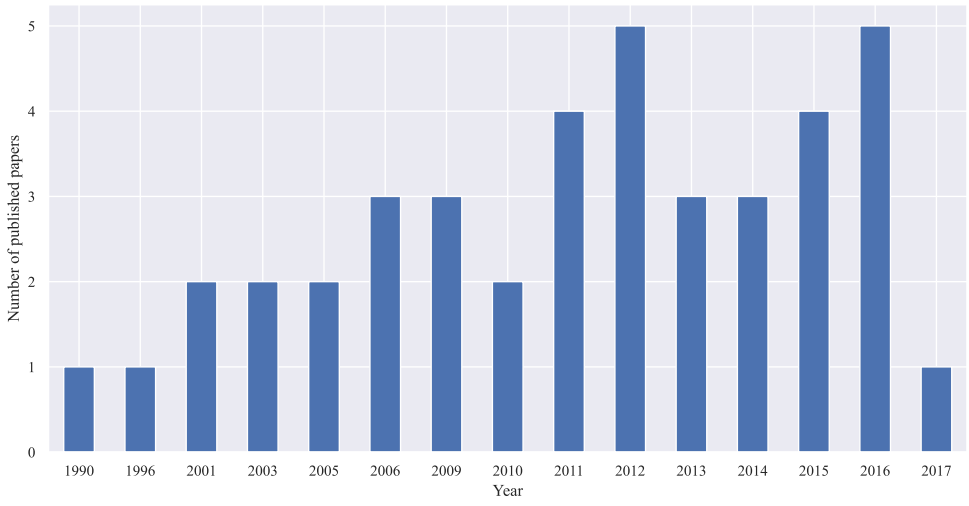

Figure A.1: *The distribution of the papers' published years [30]*

Appendix B

Case study results

B.1 Case Study A

Table B.1: *Case Study A results*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{538}	322	1.0000	3189	0.5281	0.5281	1
S_{413}	177	0.5469	5086	0.8423	0.4606	2
S_{570}	147	0.4531	5459	0.9041	0.4097	3
S_{582}	83	0.2531	6038	1.0000	0.2531	4
S_{412}	269	0.8344	1663	0.2753	0.2297	5
S_{546}	203	0.6281	1353	0.2240	0.1407	6
S_{580}	74	0.2250	3577	0.5923	0.1333	7
S_{454}	155	0.4781	1445	0.2392	0.1144	8
S_{577}	87	0.2656	2268	0.3755	0.0997	9
S_{593}	67	0.2031	2715	0.4496	0.0913	10
S_{568}	153	0.4719	795	0.1315	0.0621	11
S_{445}	156	0.4813	705	0.1166	0.0561	12
S_{735}	86	0.2625	899	0.1487	0.0390	13
S_{476}	93	0.2844	578	0.0956	0.0272	14

Continued on the next page

Table B.1: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{729}	85	0.2594	590	0.0976	0.0253	15
S_{404}	226	0.7000	205	0.0338	0.0237	16
S_{441}	180	0.5563	249	0.0411	0.0229	17
S_{600}	78	0.2375	489	0.0808	0.0192	18
S_{458}	146	0.4500	255	0.0421	0.0189	19
S_{469}	151	0.4656	217	0.0358	0.0167	20
S_{435}	167	0.5156	117	0.0192	0.0099	21
S_{581}	154	0.4750	122	0.0200	0.0095	22
S_{755}	92	0.2812	141	0.0232	0.0065	23
S_{753}	78	0.2375	157	0.0258	0.0061	24
S_{616}	105	0.3219	100	0.0164	0.0053	25
S_{503}	146	0.4500	61	0.0099	0.0045	26
S_{754}	146	0.4500	37	0.0060	0.0027	27
S_{756}	78	0.2375	61	0.0099	0.0024	28
S_{736}	146	0.4500	29	0.0046	0.0021	29
S_{716}	146	0.4500	26	0.0041	0.0019	30
S_{428}	85	0.2594	44	0.0071	0.0018	31
S_{737}	146	0.4500	23	0.0036	0.0016	32
S_{668}	86	0.2625	34	0.0055	0.0014	33
S_{675}	147	0.4531	20	0.0031	0.0014	34
S_{663}	86	0.2625	30	0.0048	0.0013	35
S_{537}	73	0.2219	35	0.0056	0.0012	36

Continued on the next page

Table B.1: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{667}	146	0.4500	15	0.0023	0.0010	37
S_{432}	167	0.5156	13	0.0020	0.0010	38
S_{618}	75	0.2281	28	0.0045	0.0010	39
S_{639}	86	0.2625	24	0.0038	0.0010	40
S_{544}	149	0.4594	14	0.0022	0.0010	41
S_{748}	92	0.2812	22	0.0035	0.0010	42
S_{536}	94	0.2875	21	0.0033	0.0010	43
S_{563}	152	0.4688	13	0.0020	0.0009	44
S_{679}	147	0.4531	13	0.0020	0.0009	45
S_{623}	86	0.2625	21	0.0033	0.0009	46
S_{578}	149	0.4594	12	0.0018	0.0008	47
S_{701}	147	0.4531	12	0.0018	0.0008	48
S_{680}	147	0.4531	12	0.0018	0.0008	48
S_{496}	156	0.4813	11	0.0017	0.0008	50
S_{757}	86	0.2625	18	0.0028	0.0007	51
S_{510}	157	0.4844	10	0.0015	0.0007	52
S_{738}	74	0.2250	19	0.0030	0.0007	53
S_{614}	85	0.2594	16	0.0025	0.0006	54
S_{468}	157	0.4844	9	0.0013	0.0006	55
S_{311}	150	0.4625	9	0.0013	0.0006	56
S_{459}	159	0.4906	8	0.0012	0.0006	57
S_{571}	72	0.2188	14	0.0022	0.0005	58

Continued on the next page

Table B.1: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{629}	149	0.4594	7	0.0010	0.0005	59
S_{547}	148	0.4562	7	0.0010	0.0005	60
S_{464}	165	0.5094	6	0.0008	0.0004	61
S_{559}	154	0.4750	6	0.0008	0.0004	62
S_{500}	86	0.2625	10	0.0015	0.0004	63
S_{742}	31	0.0906	27	0.0043	0.0004	64
S_{504}	172	0.5312	5	0.0007	0.0004	65
S_{508}	162	0.5000	5	0.0007	0.0003	66
S_{567}	152	0.4688	5	0.0007	0.0003	67
S_{566}	147	0.4531	5	0.0007	0.0003	68
S_{681}	147	0.4531	5	0.0007	0.0003	68
S_{727}	146	0.4500	5	0.0007	0.0003	70
S_{658}	74	0.2250	8	0.0012	0.0003	71
S_{659}	86	0.2625	7	0.0010	0.0003	71
S_{507}	158	0.4875	4	0.0005	0.0002	73
S_{620}	147	0.4531	4	0.0005	0.0002	74
S_{677}	146	0.4500	4	0.0005	0.0002	75
S_{740}	33	0.0969	13	0.0020	0.0002	76
S_{517}	146	0.4500	3	0.0003	0.0001	77
S_{709}	146	0.4500	3	0.0003	0.0001	77
S_{493}	163	0.5031	2	0.0002	0.0001	79
S_{506}	159	0.4906	2	0.0002	0.0001	80

Continued on the next page

Table B.1: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{467}	155	0.4781	2	0.0002	0.0001	81
S_{541}	147	0.4531	2	0.0002	0.0001	82
S_{657}	146	0.4500	2	0.0002	0.0001	83
S_{688}	146	0.4500	2	0.0002	0.0001	83
S_{694}	146	0.4500	2	0.0002	0.0001	83
S_{739}	146	0.4500	2	0.0002	0.0001	83
S_{470}	17	0.0469	7	0.0010	0.0000	87
S_{477}	82	0.2500	2	0.0002	0.0000	88
S_{583}	82	0.2500	2	0.0002	0.0000	88
S_{481}	81	0.2469	2	0.0002	0.0000	90
S_{466}	156	0.4813	1	0.0000	0.0000	91
S_{485}	2	0.0000	4	0.0005	0.0000	91
S_{732}	146	0.4500	1	0.0000	0.0000	91
S_{465}	155	0.4781	1	0.0000	0.0000	91
S_{576}	146	0.4500	1	0.0000	0.0000	91
S_{725}	146	0.4500	1	0.0000	0.0000	91
S_{711}	80	0.2437	1	0.0000	0.0000	91
S_{699}	146	0.4500	1	0.0000	0.0000	91
S_{693}	146	0.4500	1	0.0000	0.0000	91
S_{505}	159	0.4906	1	0.0000	0.0000	91
S_{676}	149	0.4594	1	0.0000	0.0000	91
S_{513}	157	0.4844	1	0.0000	0.0000	91

Continued on the next page

Table B.1: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{514}	157	0.4844	1	0.0000	0.0000	91
S_{661}	80	0.2437	1	0.0000	0.0000	91
S_{515}	166	0.5125	1	0.0000	0.0000	91
S_{520}	154	0.4750	1	0.0000	0.0000	91
S_{528}	167	0.5156	1	0.0000	0.0000	91
S_{530}	81	0.2469	1	0.0000	0.0000	91
S_{607}	148	0.4562	1	0.0000	0.0000	91
S_{543}	146	0.4500	1	0.0000	0.0000	91
S_{573}	81	0.2469	1	0.0000	0.0000	91

B.2 Case Study B

Table B.2: *Case Study B results*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{417}	297	1.0000	2502	1.0000	1.0000	1
S_{288}	224	0.6756	838	0.3347	0.2261	2
S_{258}	291	0.9733	417	0.1663	0.1619	3
S_{259}	291	0.9733	373	0.1487	0.1448	4
S_{170}	243	0.7600	373	0.1487	0.1130	5
S_{225}	209	0.6089	352	0.1403	0.0855	6
S_{72}	223	0.6711	196	0.0780	0.0523	7
S_{83}	223	0.6711	167	0.0664	0.0445	8
S_{73}	223	0.6711	164	0.0652	0.0437	9
S_{257}	231	0.7067	151	0.0600	0.0424	10
S_{268}	186	0.5067	181	0.0720	0.0365	11
S_{172}	242	0.7556	80	0.0316	0.0239	12
S_{271}	212	0.6222	89	0.0352	0.0219	13
S_{275}	282	0.9333	57	0.0224	0.0209	14
S_{305}	225	0.6800	74	0.0292	0.0198	15
S_{173}	241	0.7511	62	0.0244	0.0183	16
S_{171}	242	0.7556	60	0.0236	0.0178	17
S_{208}	240	0.7467	42	0.0164	0.0122	18
S_{190}	211	0.6178	30	0.0116	0.0072	19
S_{336}	204	0.5867	28	0.0108	0.0063	20
S_{487}	118	0.2044	78	0.0308	0.0063	21

Continued on the next page

Table B.2: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{223}	209	0.6089	17	0.0064	0.0039	22
S_{418}	118	0.2044	24	0.0092	0.0019	23
S_{224}	206	0.5956	8	0.0028	0.0017	24
S_{426}	106	0.1511	20	0.0076	0.0011	25
S_{419}	168	0.4267	7	0.0024	0.0010	26
S_{718}	74	0.0089	248	0.0988	0.0009	27
S_{421}	118	0.2044	9	0.0032	0.0007	28
S_{209}	208	0.6044	2	0.0004	0.0002	29
S_{212}	208	0.6044	2	0.0004	0.0002	29
S_{213}	206	0.5956	2	0.0004	0.0002	31
S_{343}	168	0.4267	2	0.0004	0.0002	32
S_{719}	74	0.0089	20	0.0076	0.0001	33
S_{425}	101	0.1289	2	0.0004	0.0001	34
S_{720}	74	0.0089	11	0.0040	0.0000	35
S_{423}	99	0.1200	1	0.0000	0.0000	36
S_{420}	118	0.2044	1	0.0000	0.0000	36
S_{409}	72	0.0000	945	0.3774	0.0000	36
S_{446}	118	0.2044	1	0.0000	0.0000	36
S_{344}	168	0.4267	1	0.0000	0.0000	36
S_{488}	119	0.2089	1	0.0000	0.0000	36
S_{210}	209	0.6089	1	0.0000	0.0000	36
S_{211}	209	0.6089	1	0.0000	0.0000	36

Continued on the next page

Table B.2: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{410}	72	0.0000	21	0.0080	0.0000	36

B.3 Case Study C

Table B.3: *Case Study C results*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{97}	145	1.0000	84494	1.0000	1.0000	1
S_{93}	144	0.9931	71516	0.8464	0.8405	2
S_{12}	86	0.5903	14312	0.1694	0.1000	3
S_{8}	82	0.5625	14899	0.1763	0.0992	4
S_{5}	43	0.2917	27985	0.3312	0.0966	5
S_{6}	103	0.7083	7257	0.0859	0.0608	6
S_{1}	126	0.8681	5580	0.0660	0.0573	7
S_{82}	126	0.8681	2461	0.0291	0.0253	8
S_{14}	76	0.5208	3307	0.0391	0.0204	9
S_{90}	113	0.7778	1951	0.0231	0.0180	10
S_{13}	48	0.3264	4473	0.0529	0.0173	11
S_{92}	26	0.1736	3034	0.0359	0.0062	12
S_{11}	37	0.2500	1886	0.0223	0.0056	13
S_{10}	41	0.2778	1593	0.0188	0.0052	14
S_{46}	61	0.4167	800	0.0095	0.0039	15
S_{67}	18	0.1181	1644	0.0194	0.0023	16
S_{7}	41	0.2778	594	0.0070	0.0019	17
S_{91}	95	0.6528	246	0.0029	0.0019	18
S_{70}	13	0.0833	1662	0.0197	0.0016	19
S_{95}	24	0.1597	647	0.0076	0.0012	20
S_{39}	7	0.0417	2015	0.0238	0.0010	21

Continued on the next page

Table B.3: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{96}	33	0.2222	258	0.0030	0.0007	22
S_{79}	6	0.0347	1535	0.0182	0.0006	23
S_{64}	10	0.0625	850	0.0100	0.0006	24
S_{51}	4	0.0208	2070	0.0245	0.0005	25
S_3	40	0.2708	155	0.0018	0.0005	26
S_2	14	0.0903	400	0.0047	0.0004	27
S_{15}	18	0.1181	303	0.0036	0.0004	28
S_{65}	14	0.0903	391	0.0046	0.0004	29
S_{54}	8	0.0486	477	0.0056	0.0003	30
S_{30}	5	0.0278	655	0.0077	0.0002	31
S_{58}	9	0.0556	304	0.0036	0.0002	32
S_{45}	36	0.2431	57	0.0007	0.0002	33
S_{76}	9	0.0556	214	0.0025	0.0001	34
S_{77}	10	0.0625	122	0.0014	0.0001	35
S_{80}	7	0.0417	179	0.0021	0.0001	36
S_{24}	5	0.0278	208	0.0024	0.0001	37
S_{60}	6	0.0347	142	0.0017	0.0001	38
S_{87}	4	0.0208	193	0.0023	0.0000	39
S_{63}	4	0.0208	174	0.0020	0.0000	40
S_{59}	6	0.0347	103	0.0012	0.0000	41
S_9	12	0.0764	45	0.0005	0.0000	42
S_{53}	5	0.0278	115	0.0013	0.0000	43

Continued on the next page

Table B.3: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{33}	4	0.0208	125	0.0015	0.0000	44
S_{72}	4	0.0208	117	0.0014	0.0000	45
S_{68}	4	0.0208	88	0.0010	0.0000	46
S_{17}	3	0.0139	130	0.0015	0.0000	47
S_{83}	13	0.0833	22	0.0002	0.0000	48
S_{44}	4	0.0208	82	0.0010	0.0000	49
S_{71}	4	0.0208	73	0.0009	0.0000	50
S_{85}	5	0.0278	49	0.0006	0.0000	51
S_{69}	4	0.0208	65	0.0008	0.0000	51
S_{31}	4	0.0208	53	0.0006	0.0000	53
S_{62}	3	0.0139	79	0.0009	0.0000	53
S_{78}	3	0.0139	70	0.0008	0.0000	55
S_{66}	7	0.0417	23	0.0003	0.0000	56
S_{16}	3	0.0139	57	0.0007	0.0000	57
S_{75}	6	0.0347	23	0.0003	0.0000	58
S_{29}	3	0.0139	43	0.0005	0.0000	59
S_{89}	9	0.0556	11	0.0001	0.0000	60
S_{42}	2	0.0069	51	0.0006	0.0000	61
S_{74}	2	0.0069	48	0.0006	0.0000	62
S_{84}	2	0.0069	37	0.0004	0.0000	63
S_{38}	2	0.0069	33	0.0004	0.0000	64
S_{19}	3	0.0139	15	0.0002	0.0000	65

Continued on the next page

Table B.3: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{22}	2	0.0069	24	0.0003	0.0000	66
S_{23}	3	0.0139	10	0.0001	0.0000	67
S_{47}	3	0.0139	9	0.0001	0.0000	68
S_{28}	2	0.0069	14	0.0002	0.0000	69
S_{50}	3	0.0139	6	0.0001	0.0000	70
S_{32}	2	0.0069	9	0.0001	0.0000	71
S_{36}	2	0.0069	8	0.0001	0.0000	72
S_{26}	2	0.0069	7	0.0001	0.0000	73
S_{25}	2	0.0069	3	0.0000	0.0000	74
S_{43}	2	0.0069	3	0.0000	0.0000	74
S_{61}	1	0.0000	1	0.0000	0.0000	76
S_{86}	1	0.0000	1	0.0000	0.0000	76
S_4	1	0.0000	14	0.0002	0.0000	76
S_{18}	1	0.0000	8	0.0001	0.0000	76
S_{20}	1	0.0000	10	0.0001	0.0000	76
S_{94}	1	0.0000	2	0.0000	0.0000	76
S_{21}	1	0.0000	4	0.0000	0.0000	76
S_{27}	1	0.0000	1	0.0000	0.0000	76
S_{88}	1	0.0000	8	0.0001	0.0000	76
S_{34}	1	0.0000	1	0.0000	0.0000	76
S_{35}	1	0.0000	61	0.0007	0.0000	76
S_{57}	1	0.0000	2	0.0000	0.0000	76

Continued on the next page

Table B.3: *(continued from the previous page)*

S_X	P_X	P_N	A_X	A_N	M_{PF}	P_R
S_{37}	1	0.0000	3	0.0000	0.0000	76
S_{40}	1	0.0000	1	0.0000	0.0000	76
S_{81}	1	0.0000	48	0.0006	0.0000	76
S_{41}	1	0.0000	2	0.0000	0.0000	76
S_{48}	1	0.0000	36	0.0004	0.0000	76
S_{49}	1	0.0000	26	0.0003	0.0000	76
S_{52}	1	0.0000	61	0.0007	0.0000	76
S_{55}	1	0.0000	1	0.0000	0.0000	76
S_{56}	1	0.0000	5	0.0000	0.0000	76
S_{73}	1	0.0000	3	0.0000	0.0000	76
S_{99}	1	0.0000	1	0.0000	0.0000	76