applied sciences

MDPI

Article

Fault-Prone Software Requirements Specification Detection Using Ensemble Learning for Edge/Cloud Applications

Fatin Nur Jannah Muhamad [1], Siti Hafizah Ab Hamid [1,*], Hema Subramaniam [1,*], Razailin Abdul Rashid [1] and Faisal Fahmi [2]

[1] Department of Software Engineering, Faculty of Computer Science & Information Technology, Universiti Malaya, Kuala Lumpur 50603, Malaysia; hijannahmuhamad@gmail.com (F.N.J.M.); razailin.work@gmail.com (R.A.R.)
[2] Departemen Ilmu Informasi dan Perpustakaan, Fakultas Ilmu Sosial & Ilmu Politik, Universitas Airlangga, Kampus B. Jl. Dharmawangsa Dalam, Surabaya 60286, Jawa Timur, Indonesia; faisalfahmi@fisip.unair.ac.id
* Correspondence: sitihafizah@um.edu.my (S.H.A.H.); hema@um.edu.my (H.S.)

Abstract: Ambiguous software requirements are a significant contributor to software project failure. Ambiguity in software requirements is characterized by the presence of multiple possible interpretations. As requirements documents often rely on natural language, ambiguity is a frequent challenge in industrial software construction, with the potential to result in software that fails to meet customer needs and generates issues for developers. Ambiguities arise from grammatical errors, inappropriate language use, multiple meanings, or a lack of detail. Previous studies have suggested the use of supervised machine learning for ambiguity detection, but limitations in addressing all ambiguity types and a lack of accuracy remain. In this paper, we introduce the fault-prone software requirements specification detection model (FPDM), which involves the ambiguity classification model (ACM). The ACM model identifies and selects the optimal algorithm to classify ambiguity in software requirements by employing the deep learning technique, while the FPDM model utilizes Boosting ensemble learning algorithms to detect fault-prone software requirements specifications. The ACM model achieved an accuracy of 0.9907, while the FPDM model achieved an accuracy of 0.9750. To validate the results, a case study was conducted to detect fault-prone software requirements specifications for 30 edge/cloud applications, as edge/cloud-based applications are becoming crucial and significant in the current digital world.

Keywords: requirement engineering; software requirements specification; natural language processing; ambiguity; fault-prone detection; boosting and edge/cloud applications

Citation: Muhamad, F.N.J.; Ab Hamid, S.H.; Subramaniam, H.; Abdul Rashid, R.; Fahmi, F. Fault-Prone Software Requirements Specification Detection Using Ensemble Learning for Edge/Cloud Applications. *Appl. Sci.* **2023**, *13*, 8368. https://doi.org/10.3390/app13148368

Academic Editor: Paolino Di Felice

Received: 19 May 2023
Revised: 14 June 2023
Accepted: 19 June 2023
Published: 19 July 2023

1. Introduction

Cloud computing has experienced a surge in popularity in recent years as an increasingly preferred option for deploying software systems [1]. However, to ensure that these software systems meet the requirements, a clear and well-defined software requirements specification (SRS) that outlines the functional and non-functional requirements of the system is required. In recent years, software developers and requirement engineers have faced challenges of low efficiency and poor performance in IT projects due to poorly written requirements, resulting in 82% of reworked applications being attributed to requirement errors. Consequently, companies will allocate more than 41.5% of their new project development resources towards addressing superfluous or poorly described requirements [2]. Around 87.7% of software requirements documentation is created using natural language, which results in ambiguity and leads to different interpretations, causing rework, higher maintenance expenses, and delays in software projects [3]. Due to the inherent nature of NL and the involvement of multiple stakeholders, requirements are prone to redundancy, inconsistency, and ambiguity, often referred to as software faults [4]. Addressing ambiguity

in software requirements helps developers understand and interpret the desired functionality clearly, which reduces misunderstandings and errors during development. Researchers frequently highlight the existence of different types of linguistic ambiguity, such as lexical, syntactic, semantic, syntax, and pragmatic. However, the existing proposed techniques fail to sufficiently tackle all forms of linguistic ambiguity, resulting in inadequate accuracy in detecting specific types of ambiguity.

Previous studies have introduced different approaches for detecting linguistic ambiguity in software requirements. The ambiguity detector works as an algorithm to classify ambiguities but is limited to lexical, syntactic, or syntax ambiguity [5], while Sabriye and Zainon [6] also implemented the same ambiguity detector but only for syntactic and syntax ambiguity in SRS documents. Hence, Rani and Aggarwal [7] improved this approach by adding referential ambiguity detection and lexical, syntactic, syntax, and pragmatic ambiguity detection, but their approach did not cover semantic ambiguity. Bajwa et al. [8] proposed NL2OCL, which translates natural language software constraints into formal constraints. Ferrari et al. [9] proposed a natural language processing approach based on Wikipedia crawling and word embedding to detect domain-specific ambiguities. Osman and Zaharin [10] proposed Ambi-Detect, which detects ambiguity in Malay SRS documents using Random Forest to classify the ambiguous and unambiguous software requirements; however, Malay SRS is only a small dataset. The ChatGPT system, which was developed by OpenAI and utilizes the GPT-4 architecture, is capable of identifying ambiguous software requirements. However, ChatGPT does not take into account other essential components of the SRS, such as the title, description, and intended users.

In this paper, we focus on detecting whether the SRS is prone to fault or a clean SRS by proposing the fault-prone SRS detection model that involves the ambiguity classification model. The ambiguity classification model classifies software requirements based on five major linguistic ambiguities (lexical, syntactic, semantic, syntax, and pragmatic ambiguity) using deep learning techniques. Hence, the fault-prone software requirements specification model exploited the boosting algorithms, which are Adaptive Boosting, Gradient Boosting, and Extreme Gradient Boosting, to improve the model's accuracy to detect the fault-prone SRS based on the titles, descriptions, presence of users, and classified ambiguous software requirements of the SRS. The dataset highlighted the edge/cloud application SRS and covered a wide range of topics to assure the model was trained adequately. Hence, to aid the detection of fault-prone SRS, we propose the fault-prone severity scale in our case study, which is derived from key components of the SRS, including the title, description, intended users, and software requirements. The scale categorizes ambiguity as low, moderate, or high, based on a calculated score. The research instrument used is quantitative research by carrying out quasi experimental research by labeling the data based on the type of ambiguity or clean requirements. The contributions of this paper are as follows:

- Development of a fault-prone software requirements specification detection model to ensure high accuracy in detecting the fault-prone SRS by utilizing the ambiguity classification model on ambiguous software requirements, title, description, and intended users of the SRS.
- Analysis of the fault-prone SRS of edge/cloud applications to reduce and identify potential issues early in the development process, allowing developers to make necessary changes and adjustments to ensure the application meets the needs of its users.

The remainder of the paper is structured as follows: Section 2 outlines the research background; Section 3 gives an overview of related topics; Section 4 presents the proposed models to detect the ambiguity of software requirements in SRS; Section 5 illustrates the case study in this research; and Section 6 discloses the conclusion and future plans of this research.

2. Research Background

2.1. Software Requirements Specification

A software requirements specification is a document that outlines the functional requirements (FR), non-functional requirements (NFR), and constraints for a software system. FR describes the system's functionality, while NFR describes the system's properties and constraints [11]. The creation of SRS documents during the early stages of software development serves as a key reference for all stakeholders involved in the project, including developers, testers, and project managers. Wrong or missing requirements lead to wrong or incomplete products, regardless of how good the subsequent phases are [12]. According to a study by James Martin, 50% of all requirement defects stem from poorly written, unclear, ambiguous, or inaccurate requirements, while the other 50% are caused by inadequate specifications, such as incomplete or missing requirements. However, despite these statistics, a staggering 70% of organizations fail to take practical measures to enhance their requirements' quality [2]. (Figure 1)

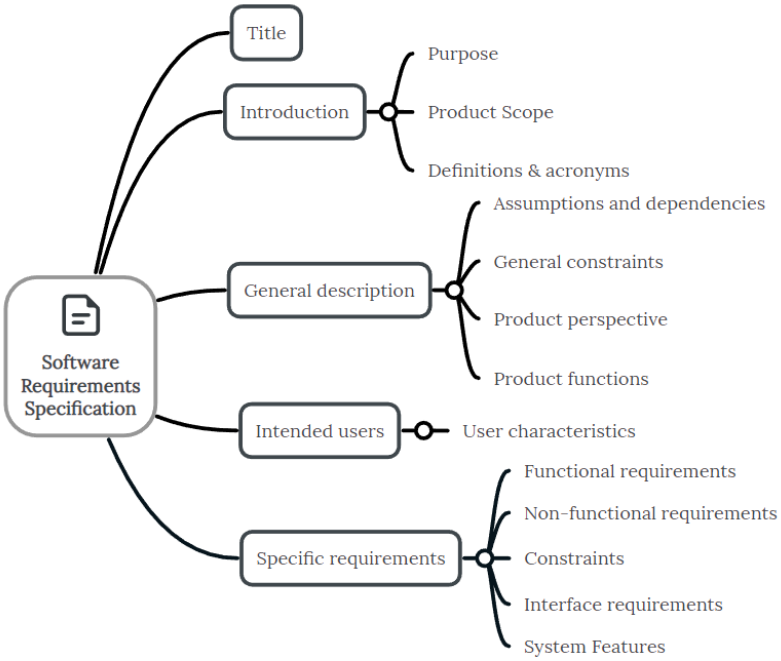


Figure 1. Software Requirements Specification.

2.2. Fault-Prone Software Requirements Specification

A fault is an unappealing or undesirable aspect, especially in a piece of work. Software faults are errors, flaws, or the failure of computer programs. In requirement engineering, a fault is a manifestation of missing, incorrect, or ambiguous information [13]. Fault-prone in requirement engineering indicates the extent to which a requirement is prone to faults, thereby causing software failure. Fault-prone SRS can have a significant impact on software project construction that can extend to subsequent phases of the requirement engineering process. Conversely, a clean SRS means that most of the requirements are reliable and understood by stakeholders, project teams, developers, and users. Other than considering software requirements in detecting fault-prone SRS, considering the title, description, and intended users is essential, as these factors provide vital information about the software system.

According to the IEEE Standard for Software Requirements Specifications [14], the title and description of an SRS should provide a clear and concise overview of the software system and its main functionalities. Studies show that the quality of SRS documents, including the title and description, significantly impacts the fault-proneness of the software system [15]. If the title and description suggest the system has a high degree of complexity or involves multiple subsystems, this may indicate that the system is more prone to faults

and requires thorough testing. Conversely, if the title and description are clear and concise and the system is relatively simple, this may indicate that the system is less prone to faults and may require less testing. The standard also recommends that the SRS include information on the intended users and the expected behavior of the system. A study by Aggarwal et al. [16] found that the intended user plays a crucial role in determining the quality of the software system. Table 1 depicts the factors contributing to the fault-prone SRS.

Table 1. Other factors contribute to fault-prone SRS other than software requirements.

Factor	Title	Description	Intended Users	Explanation
Title	ProDash	ProDash is a software application that helps project managers track and manage projects. The system will provide a centralized platform for project managers to monitor project progress, assign tasks, and collaborate with team members. The dashboard will display real-time project data and analytics, allowing managers to make informed decisions about project timelines and resource allocation.	• Project managers • Team leaders Executives The system will be particularly useful for teams working on complex projects with multiple stakeholders and dependencies.	A vague or confusing title leads to miscommunication and misunderstandings among stakeholders. The title of a project is an essential component of its overall description and should be clear and concise. The project title should accurately reflect the purpose of the project and provide a clear indication of what the project aims to accomplish. Having a clear title helps people who are involved understand the project's purpose and focus.
Description	Virtual Event Management System	A software application designed to help businesses and individuals plan, organize, and manage events of all types and sizes. The system provides a range of tools for creating and managing events, including event scheduling, budget management, vendor and attendee management, and task tracking. The system enables event planners to create and manage event calendars. The system is intended to streamline the event planning process and provide real-time analytics and reporting.	• Event planners • Marketing professionals • Businesses looking to host virtual events. The system is particularly useful for those who want to engage with a large audience remotely, such as in-person event organizers who want to transition to virtual events due to COVID-19.	The system serves different purposes and is designed for different types of events. Based on the title, we expect it to be specifically designed to help organizations host and manage virtual events, providing features such as online registration, virtual venue setup, and live streaming capabilities. Plus, there is an emphasis due to COVID-19.
Intended Users	Veritas Student Portal	Veritas Student Portal is a web-based platform that serves as the primary online resource for students, faculty, staff, and prospective students. The website provides a range of information and services, including course catalogs, event calendars, news and announcements, academic and financial aid resources, and access to online learning platforms.	• Students • Faculty and Staff • Prospective Students • Alumni • Public	Intended users for the Veritas Student Portal suddenly expanded to include alumni and the public, it would be important to update the SRS accordingly. This is because the needs and requirements of alumni and the public may differ significantly from those of students, faculty, and staff. The clear presence of users in the SRS is important because it helps software developers and stakeholders to understand the specific needs and requirements of each user group.

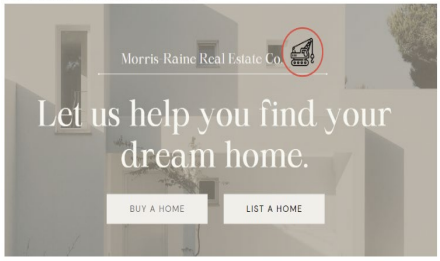
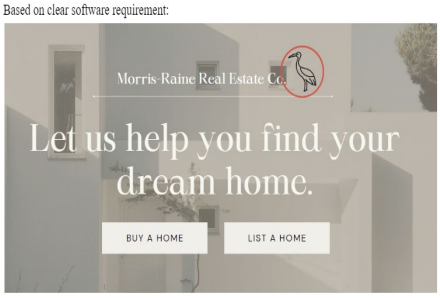
2.3. Ambiguity in Software Requirements Specification

Ambiguity is a common problem in natural language that arises due to a variety of factors, such as grammatical errors, word choice, a lack of detail, and multiple meanings. This study is primarily focused on language ambiguity within the context of software requirements engineering. Understanding the various types of ambiguity that arise in the SRS is crucial to minimizing errors and improving the overall quality of the SRS. A study conducted by Sandhu and Sikka identified five different types of natural language SRS ambiguity. These include lexical ambiguity, syntactic or structural ambiguity, semantic or scope ambiguity, pragmatic ambiguity, and syntax ambiguity. Understanding these different types of ambiguity is essential to effectively communicating requirements and ensuring that the SRS is free from ambiguity, thereby reducing the potential for errors and misunderstandings [6]. Table 2 shows examples of ambiguous software requirements that lead to incorrect development.

Table 2. Examples of ambiguous software requirements that lead to incorrect development.

Ambiguous Software Requirement	Detected Ambiguity	Program Code/ User Interface	Explanation	Recommended Software Requirement

The system shall be able to easily navigate to the main features of the app.	Syntactic	Based on ambiguous software requirement:
		class ClickCounter:
		# Example usage:
		counter = ClickCounter()
		button1 = Button(text="Feature 1",
		command=lambda: counter.count_click())
		
		Based on clear software requirement:
		class ClickCounter:
		def __init__(self):
		self.clicks = 0
		def count_click(self):
		self.clicks += 1
		if self.clicks > 3:
		raise ValueError("Exceeded maximum number of clicks")
		# Example usage:
		counter = ClickCounter()
		button1 = Button(text="Feature 1",
		command=lambda: counter.count_click())
			The ambiguous software requirement lacks specific details about how navigation should be achieved and does not provide any information about the number of taps or clicks required to reach the main features, or the specific navigation methods to be used. People may interpret this to mean that navigation should be achieved through gestures, voice commands, or other methods. Additionally, it does not provide any specific criteria for what constitutes "easy" navigation. This lack of clarity can result in different interpretations and expectations for the final product.	The system shall be able to easily navigate to the main features of the app using not more than three taps or clicks.

The system shall display the company crane logo on the home page.	Lexical	Based on ambiguous software requirement:  Based on clear software requirement:  	This ambiguous software requirement is not specific. This is because "Crane" refers to a bird with long legs and a long neck. Additionally, "Crane" also refers to a large machine used for lifting and moving heavy objects.	The system shall display the company crane (bird) logo on the home page.

2.4. Fault-Prone Severity Scale

The proposed fault-prone severity scale aids in the detection of fault-prone SRS. The scale is based on a calculated weighted score derived from key components of the SRS document, including the title, description, intended users, and software requirements. The purpose of this scale is to identify and categorize the level of ambiguity present in the SRS document, with the goal of improving the reliability of the software system being developed. Each component is evaluated and assigned a score based on the level of ambiguity present. For instance, the title and description are evaluated based on the level of detail provided and the clarity of the language used. The intended user component is evaluated based on the presence and specificity of the user information provided, while the software requirements are evaluated based on the presence of ambiguous words in the software requirements. This scale categorizes the level of ambiguity present as either low, moderate, or high, with high levels of ambiguity indicating a higher likelihood of faults and errors in the software system being developed. A detailed discussion regarding the proposed fault-prone severity scale is presented in Section 5.

3. Related Studies

Nigam et al. [5] proposed an ambiguity detector tool for assessing requirements specifications by identifying lexical, syntactic, and semantic ambiguities in the requirements. The input of the ambiguity detector is requirements specifications and the corpus. Then, the requirements are processed with a Stanford POS tagger and an algorithm is implemented for detecting each ambiguous sentence. Four SRS documents with different numbers of lines were evaluated to determine the existence of ambiguities in each document. The percentages of the detected ambiguities are shown for each ambiguity. Sabriye and Zainon [6] also implemented the same ambiguity detector for syntactic and syntax ambiguity in SRS documents. The researchers developed a prototype tool to evaluate the proposed approach. However, the study extracts a very limited dataset for the development of the tool, and only displays the detected ambiguity without saving the changes made on the detected ambiguity. Rani and Aggarwal [7] improved this approach by adding referential ambiguity detection. Thus, the researchers tend to present lexical, syntactic, syntax, and pragmatic ambiguity detection in their work; however, only seven requirements were evaluated as a dataset. Nonetheless, all these studies do not perform any performance evaluation for the proposed approach to identify the accuracy of the performed algorithm in detecting ambiguities.

In 2017, Ferrari et al. [9] proposed a natural language processing (NLP) approach to detect domain-specific ambiguities in computer science terminology. The method involves crawling Wikipedia to extract both computer science (CS) and domain-specific documents and pre-process them. Next, the most frequently occurring nouns in CS documents are ranked, and a modified form of each noun is injected into domain-specific documents. The word2vec method is used to train word embeddings on a corpus of CS and domain-specific documents. Finally, the similarity of embeddings for CS nouns and the modified variants in domain papers is compared, which estimates the variance in meaning of CS nouns when used in different domains. This approach focuses solely on nouns and shows promising results in preliminary studies on five domains. Further validation of this method is needed to avoid misunderstandings when modifying documents and engaging with specialists in relevant disciplines.

Bäumer and Geierhos [17] described a method for constructing a software system that integrates existing expert tools and controls them using automated compensation algorithms to help end-users create unambiguous and complete requirements specifications. The complete text analysis pipeline is built ad hoc and, hence, is adapted to the specific conditions of a requirements description based on ambiguity indicators. The purpose of this method is to detect ambiguity and incompleteness in natural language software requirements and automatically correct them. This approach not only covers pragmatic ambiguity but also incompleteness, lexical, and syntactic ambiguities. The evaluation result of the indicator quality for incompleteness is 0.73 for accuracy, 0.70 for recall, 0.83 for precision, and 0.72 for F-Score. For referential ambiguity, it scored 0.82 for accuracy, 0.710 for recall, 0.93 for precision, and 0.75 for F-Score. For syntactic ambiguity, it scored 0.80 for accuracy, 0.71 for recall, 0.87 for precision, and 0.74 for F-Score. However, the pragmatic ambiguity covered in this approach is limited to referential ambiguity.

Osama and Aref [18] proposed DARA, a method for detecting and resolving ambiguity in SRS. The lexical, referential, coordination, scope, and vague domains are the focus of the tool. DARA also used the provided approach by Ayan et al., 2012 [5] in the ambiguity detection design. As a result, the authors highlighted the indicators for detecting each ambiguity handled using a rule-based approach. A total of 36 sets with a different number of requirements in various SRS domains were gathered from various sources. DARA examines the number of detected sentences, number of resolved sentences, and time spent for each SRS that undergoes ambiguity detection in addition to calculating the percentages for each ambiguity found as indicators. The results of utilizing DARA on these 36 case studies reveal that potential ambiguities occur frequently, accounting for almost 60% of the total number of requirements sentences (lexical ambiguity 37%, referential ambiguity 9%,

coordination ambiguity 13%, scope ambiguity 25%, and vague 16%). DARA resolves 67% of ambiguity in the total number of required sentences. However, the performance of the proposed approach was not evaluated.

Osman and Zaharin [10] proposed an automated approach, Ambi-Detect, for detecting ambiguities in Malay SRS documents, which consist of 180 manually labeled requirements, and used supervised machine learning methods, such as Random Forest, to classify the ambiguous and unambiguous software requirements. The classification result achieved an accuracy score of 89.67%, precision score of 0.90, recall score of 0.88, and F-Measure of 0.89, which is reasonably acceptable and may improve the productivity of formulating SRS. The proposed automated approach is a valuable tool for improving the quality of SRS documents and reducing the potential for misunderstandings and errors in Malay SRS. Nevertheless, this study uses a small dataset that affects the accuracy of the performance, thus making the classification arguable. Hence, the researchers have not stated which classification of ambiguity is covered in the research. However, further research is needed to evaluate the approach on a larger dataset.

4. Proposed Fault-Prone Software Requirements Specification Detection Model

The study aims to detect fault prone software requirements specifications (SRS) due to the ambiguous requirements based on classified language ambiguity, the clarity of the title, description, and the presence of intended users. The development of the fault-prone software requirements specification detection model (FPDM) comprises the ambiguity classification model, as depicted in Figure 2. We used Python as the main language to develop the predictive model. Our machine for model development was a Vivo book Asus Laptop with a 3.20 GHz AMD Ryzen and 16 GB RAM from Pro System Machine Sdn. Bhd. Kuala Lumpur, Malaysia. The operating system used was Windows 11 with 21H2 version, which ran on 5800H with Radeon Graphics. The model covered five types of linguistic ambiguity: lexical ambiguity, syntactic ambiguity, semantic ambiguity, syntax ambiguity, and pragmatic ambiguity.

4.1. Ambiguous Classification Model

4.1.1. Phase 1: Data Collection

The first phase in constructing ACM is data collection. This dataset covers a wide spectrum of topics to ensure the model is trained thoroughly. The process involves extracting the requirements (including functional requirements and non-functional requirements). This is a self-collected dataset on online search engine sources, and we managed to collect 100 sets of SRS. One of the sources of the collected SRS is from the existing research by Osama and Aref [18] that presents 36 SRS. The collected requirements are then stored in csv format.

4.1.2. Phase 2: Data Processing

After collecting the data, we cleaned the data by applying Python code to remove stop words, non-alphanumeric characters, duplicated requirements, and word stemming, and transformed each word to lowercase. Data cleaning is crucial to ensuring that the requirements are readable and easy to process. We have 7061 software requirements based on 100 SRS (Figure 3).

- Sentence splitter: The sentence splitter function isolates each sentence from the input text and turns the sentence into individual sentences.
- Tokenizer: The tokenizer function takes each sentence as input and breaks the sentence down into tokens, such as words, numbers, and punctuation.
- NLTK Part of speech Tagging (POS tagger): The parts of speech (POS) tagger function is the process of marking up words in text format for a specific segment of a speech depending on the definition and context.
- Syntactic parser: The syntactic parser function converts sequences of words into structures that reveal how the parts of a sentence are interconnected.

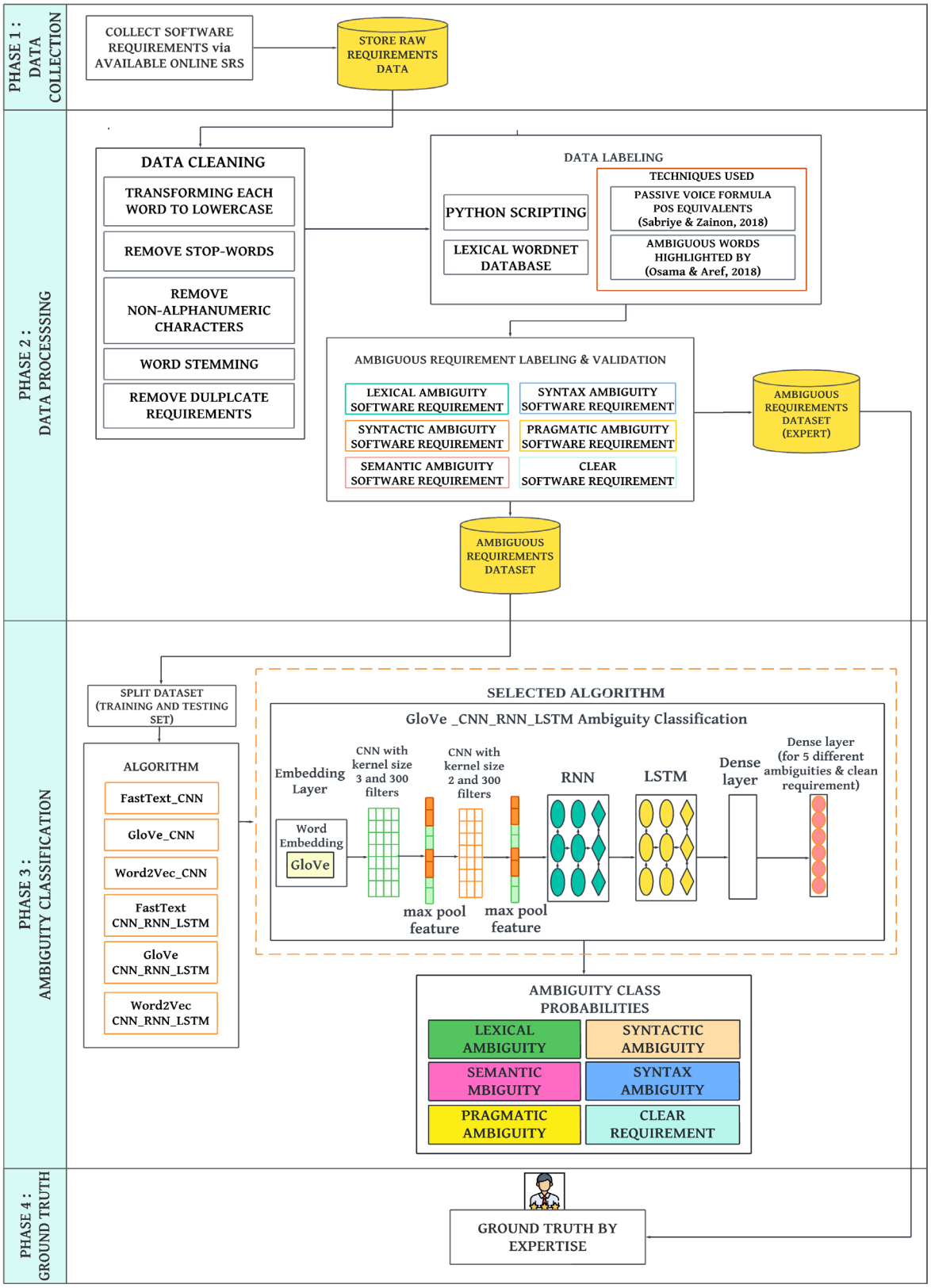


Figure 2. Ambiguity Classification Model [6,18].

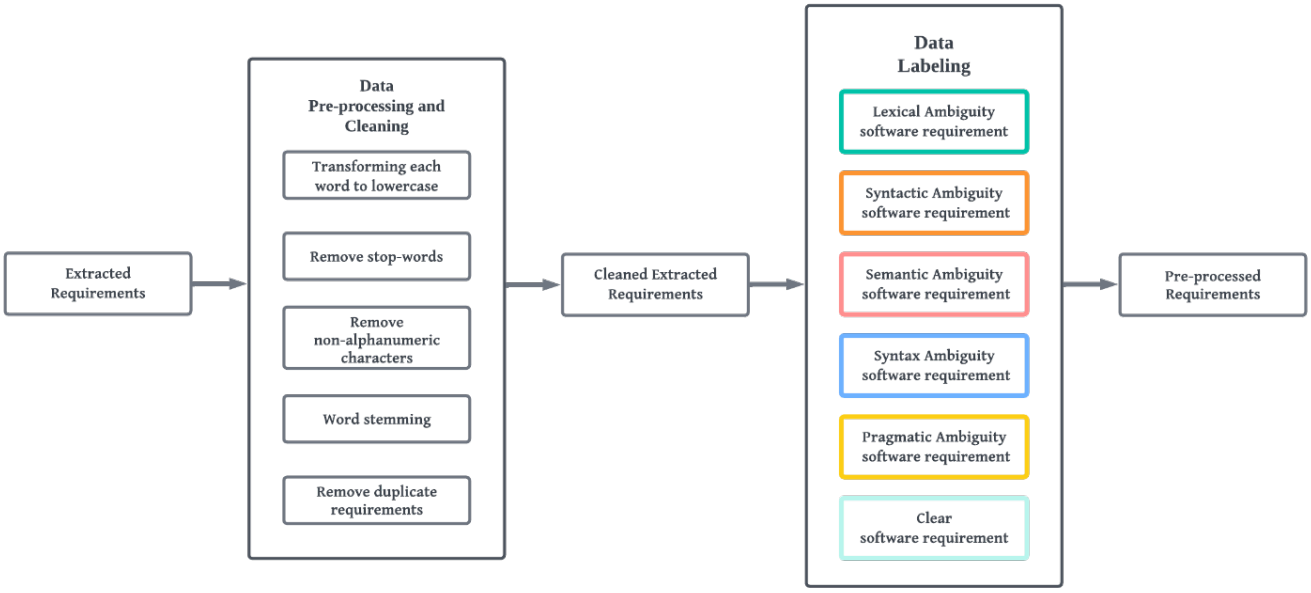


Figure 3. Data cleaning and pre-processing.

4.1.3. Data Labelling

The next process involves data labeling, which aims to categorize the gathered requirements into six groups, namely lexical ambiguity, syntactic ambiguity, semantic ambiguity, syntax ambiguity, pragmatic ambiguity, and clean. To expedite this process and eliminate the need for human labeling, Python scripts were developed utilizing algorithms to identify and differentiate between the various types of ambiguities. The POS tagger was employed to assign grammatical information to each word in the sentence, while Osama and Aref [18] highlighted ambiguous words were used to identify each type of ambiguous requirement, as illustrated in Table 3. Furthermore, the passive voice formulas and POS equivalents, based on the research by Sabriye and Zainon [6], were employed to determine ambiguous requirements. The lexical database WordNet was also utilized to facilitate ambiguity detection in software requirements. WordNet offers a comprehensive inventory of words and semantic relationships, enabling us to identify potential sources of ambiguity in requirement statements by analyzing constituent terms and associated synonyms and senses. By leveraging WordNet's extensive coverage of the English language lexicon, we developed a robust approach to automatically detect and classify different types of ambiguities in software requirements. This process resulted in 7061 extracted requirements, with 3041 labeled as ambiguous requirements and 4020 labeled as clean requirements, as presented in Table 4.

Table 3. Possible Ambiguity Indicator.

Ambiguity	Possible Ambiguity Indicators
Lexical	Lexical ambiguity indicators: Access, address, application, archive, array, bandwidth, binary, cache, compiler, compression, configuration, console, data, directory, disk, domain, driver, encryption, file, firewall, folder, gateway, interface, kernel, library, link, load, logic, macro, malware, memory, metadata, migration, monitor, object, optimization, packet, path, pixel, protocol, query, registry, resource, router, script, security, server, etc.
Syntactic	Syntactic ambiguity indicators: And, or, but, so, yet, nor, for, if, although, because, since, unless, until, while, even though, then, as, whenever, wherever, whereas, as if, as long as, etc.
Semantic	Semantic ambiguity indicators: All, every, many, several, any, some, few, a lot of, much, little, enough, most, none, half, whole, both, either, neither each, more, less, plenty of, a number of, a great deal of, a bit of, a few, a majority of, etc.
Syntax	Checks the absence of a full stop at the end of a condemnation, indicated by the "./." tag, or the use of passive voice for each software requirement.
Pragmatic	Pragmatic ambiguity indicators: I, me, you, he, him, she, her, it, we, us, they, them, mine, yours, his, hers, its, ours, theirs, myself, yourself, himself, herself, itself, ourselves, yourselves, themselves, this, that, these, those, somebody, someone, something, etc.

Table 4. Number of tweets in the dataset.

Software Requirements	Labelled Dataset
Lexical Ambiguity	474
Syntactic Ambiguity	615
Semantic Ambiguity	614
Syntax Ambiguity	504
Pragmatic Ambiguity	834
Clean	4020
Total	**7061**

4.1.4. Phase 3: Ambiguity Classification

ACM uses deep learning algorithms to ensure high accuracy in detecting and classifying different types of ambiguities or clean requirements. To develop the ACM, we started with a pre-processed Ambiguous Requirements Dataset containing 7061 software requirements, which was then split into training and testing sets at an 80/20 ratio. We utilized two deep learning algorithms, Convolutional Neural Network (CNN), and a combination of CNN, Recurrent Neural Networks (RNNs), and Long Short-Term Memory networks (LSTMs). We also employed three word embedding techniques: GloVe, Word2Vec, and FastText, to convert words into vector representations. GloVe uses global co-occurrence statistics to capture semantic relationships between words, identifying ambiguities from multiple meanings [19]. Word2Vec provides vector representations of text, capturing semantic meaning for NLP tasks [20], while FastText extends Word2Vec with a shallow neural network, capturing context for a fine-grained understanding of text [21]. We stacked the algorithms as Word2Vec_CNN, GloVe_CNN, FastText_CNN, Word2Vec_CNN_RNN_LSTM, GloVe_CNN_RNN_LSTM, and FastText_CNN_RNN_LSTM. Table 5 shows presents the details on each layer for each experiment for the proposed model.

Table 5. Details on each layer in the proposed model for each experiment.

Algorithm	Main Layer	Neural Network Layer	Output Size	Kernel Size	Max-Pool Size	Activation Function
Word2Vec_CNN	Input layer	Word2Vec Embedding Layer	300	None	None	None
	Hidden	CNN Layer 1	300	3	50	ReLU
	layer	CNN Layer 2	300	2	10	ReLU
	Fully	Dense Layer 1	300	None	None	ReLU
	connected layer	Dense Layer 2 (output layer)	6	None	None	None
GloVe_CNN	Input layer	GloVe Embedding layer	300	None	None	None
	Hidden	CNN Layer 1	300	3	50	ReLU
	layer	CNN Layer 2	300	2	10	ReLU
	Fully	Dense Layer 1	300	None	None	ReLu
	connected layer	Dense Layer 2 (output layer)	6	None	None	None
FastText_CNN	Input layer	FastText Embedding Layer	300	None	None	None
	Hidden	CNN Layer 1	300	3	50	ReLU
	layer	CNN Layer 2	300	2	10	ReLU
	Fully	Dense Layer 1	300	None	None	ReLU
	connected layer	Dense Layer 2 (output layer)	6	None	None	Softmax
Word2Vec_CNN_ RNN_LSTM	Input layer	Word2Vec Embedding Layer	300	None	None	None
	Hidden	CNN Layer 1	300	3	50	ReLU
	layer	CNN Layer 2	300	2	10	ReLU
		RNN Layer 1	300	None	None	Sigmoid
		LSTM Layer 1	256			ReLU
	Fully	Dense Layer 1	300	None	None	ReLU
	connected layer	Dense Layer 2 (output layer)	6	None	None	None

Table 5. *Cont.*

Algorithm	Main Layer	Neural Network Layer	Output Size	Kernel Size	Max-Pool Size	Activation Function
GloVe_CNN_ RNN_LSTM	Input layer	GloVe Embedding Layer	300	None	None	None
	Hidden layer	CNN Layer 1	300	3	50	ReLu
		CNN Layer 2	300	3	10	ReLu
		RNN Layer 1	300	2	None	ReLu
		LSTM Layer 1	128			
		Dense Layer 1	300	None	None	ReLu
	Fully connected layer	Dense Layer 2 (output layer)	6	None	None	Softmax
FastText_CNN_ RNN_LSTM	Input layer	FastText Embedding Layer	300	None	None	None
	Hidden layer	CNN Layer 1	300	3	50	ReLu
		CNN Layer 2	300	3	10	ReLu
		RNN Layer 1	300	2	None	ReLu
		LSTM Layer 1	128			
	Fully connected layer	Dense Layer 1	300	None	None	ReLu
		Dense Layer 2 (output layer)	6	None	None	Softmax

Using the prepared datasets for the ACM, we present the training results of the algorithms for the ACM in Table 6, based on its performance in the ambiguity classification of software requirements. To calculate the ambiguity for each requirement in the ACM, each requirement was separated into a single word based on the type of ambiguity. The table presents the evaluation metrics, including accuracy, recall, precision, and F-measure for each model. Figure 4 illustrates the accuracy of the algorithms. Based on the results, the GloVe_CNN_RNN_LSTM model achieved the highest score of 0.9907 for accuracy. The F-measure score, which shows the balance of precision and recall, 0.9664 and 0.9598 respectively, was also high, indicating the model's ability to balance the classification of ambiguities. Therefore, the GloVe_CNN_RNN_LSTM model is considered the highest-performing model and will be chosen for the next stage of the project.

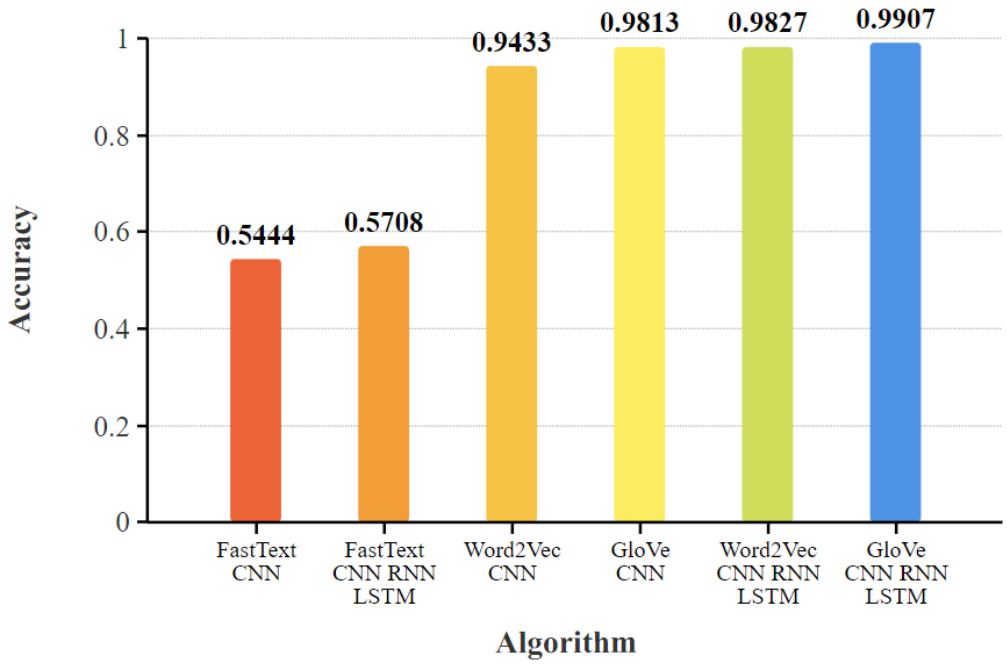


Figure 4. Accuracy of algorithms in creating ACM.

Table 6. Training results for each algorithm in ACM.

Algorithm	Precision	Recall	F-Measure
FastText_CNN	0.9366	0.7636	0.8413
GloVe_CNN	0.9641	**0.9553**	0.9597
Word2Vec_CNN	0.9561	0.9156	0.9356
FastText_CNN_RNN_LSTM	0.5712	0.5713	0.5712
GloVe_CNN_RNN_LSTM	**0.9664**	0.9533	**0.9598**
Word2Vec_CNN_RNN_LSTM	0.9594	0.9509	0.9551

The proposed GloVe_CNN_RNN_LSTM algorithm utilizes a sequential neural network architecture comprising an embedding layer with parameters such as max features and embedding size. A dropout layer is also applied to remove some context from the input dataset. Conv1D and MaxPooling layers are then added to the architecture to extract higher-level features and expedite the process. Additionally, RNN and LSTM layers are employed to capture long-term dependencies between word sequences and grasp the context of the input sentence. To enhance the efficiency of the hybrid model, dropout and dense layers are added in the final stages. Two activation functions, namely Relu in Conv-1D and Softmax in dense layers, are used. The suggested model is compiled with the optimizer Adam.

Table 7 shows the performance of the GloVe_CNN_RNN_LSTM algorithm for each class of ambiguity in the software requirements. The algorithm achieved a high precision score of 0.9860, 0.9826, and 0.9739 for pragmatic, lexical, and syntactic ambiguities, respectively. The recall score, which measures the true positive rate of the correctly classified instances, was also high for all ambiguity classes, ranging from 0.9621 to 0.0.9848. The f-measure, which is the harmonic mean of precision and recall, was also high for all classes, with a minimum value of 0.9628 for semantic ambiguity and a maximum value of 0.9854 for pragmatic ambiguity. These results indicate that the GloVe_CNN_RNN_LSTM algorithm is effective in classifying different types of ambiguity in software requirements, with a high level of accuracy, recall, and f-measure for each ambiguity class. Therefore, we utilized this algorithm as the basis for developing a reliable and effective ambiguity classification model for software requirements.

Table 7. Performance for each ambiguity class in ACM.

Algorithm	Ambiguity	Precision	Recall	F-Measure
GloVe_CNN_RNN_LSTM	Lexical	0.9826	0.9811	0.9819
	Syntactic	0.9739	0.9745	0.9742
	Semantic	0.9634	0.9621	0.9628
	Syntax	0.9674	0.9660	0.9667
	Pragmatic	0.9860	0.9848	0.9854

An accuracy of 0.9907 is considered reliable as it combines advanced techniques such as GloVe embeddings, CNN, RNN, LSTM, and expert validation. These techniques enable the algorithm to effectively classify and detect ambiguity in software requirements by capturing patterns and nuances in the data. The involvement of subject matter experts adds credibility to the accuracy score, ensuring alignment with their domain knowledge.

4.1.5. Phase 4: Ground Truth

This process ensures that the model accurately classifies software requirements and helps identify potential sources of ambiguity in software projects. To achieve accuracy, we collected another 10 SRS, which corresponded to 254 requirements that were available online. We invited subject matter experts to validate and evaluate our initial approach of detecting an ambiguous requirement. Five respondents (e.g., working as system analysts and/or having a background in computer science with requirement engineering knowledge) were selected. Two of the respondents had at least five years of experience, and three respondents had two to three years of experience in requirements engineering. In total, 254 software requirements were divided among the five experts. The experts were provided with a questionnaire with 50 or 54 requirements, focusing on labeling the

ambiguous or clear requirements based on knowledge and understanding. Additionally, we provided a brief description of each type of ambiguity to assist the experts in making their determinations. The experts were given 14 days to provide feedback, allowing them to review the requirements at their convenience and accurately identify any ambiguous or clear requirements. Based on the result from the questionnaire:

- One respondent provided the same result as the one generated by the ACM.
- Three respondents did not agree with 5 out of 50 software requirements that were detected as ambiguous, but the experts identified the software requirement as clear.
- One respondent did not agree with 6 out of 50 software requirements that were detected as ambiguous, but he identified the software requirement as clear.
- On average, across all five experts, only 8.27% of the responses concerning ambiguous and clean requirements did not match the results provided by ACM.

Based on the success of the results from the ambiguity classification model, we proceeded with our experiments for the fault-prone software requirements specification detection model.

4.2. Fault-Prone Software Requirements Specification Detection Model

For FPDM development, we experimented with different boosting ensemble learning algorithms for the model, including adaptive boosting (AdaBoost), gradient boosting (GBM), and extreme gradient boosting (XGBoost), and selected the best algorithm as the classifier for the model. The boosting algorithm was one of the many elements used in machine learning for the creation of a predictive model that utilized the data frame prepared in Python. Consequently, the data frame fed into different machine learning elements that utilized the boosting approach to build the predictive models. Boosting is one of many machine learning elements used to construct a predictive model that makes use of the Python data frame. To ensure unbiased data distribution, the training and testing sets use the 80/20 percent splitting technique before randomly picking the data and evenly dividing them according to class.

4.2.1. Phase 1: Data Collection

The creation of the fault-prone software requirements specification dataset facilitates the development of the fault-prone software requirements specification detection model. The dataset comprises 200 software requirements specifications (SRS) that are publicly accessible online. This includes SRS title, SRS description, SRS intended users, and SRS requirements for each SRS.

4.2.2. Phase 2: Data Processing

Before the dataset is utilized for training the FPDM, the data undergo a process of cleaning and pre-processing utilizing the same method employed in the ambiguity classification model. This involves the elimination of redundant information, formatting of the text, and ensuring the uniformity of data across all documents. To determine the quality of the SRS, several criteria were employed.

- SRS Title: The SRS title was evaluated to ensure that the title provides a clear idea of the system that will be implemented.
- SRS Description: The frequency of words present in the SRS description was analyzed by counting each word from the SRS title in the SRS description. To ensure consistency, the titles were converted into root word before calculating the frequency of words. The Computer Science Academic Vocabulary List (CSAVL) was excluded to ensure that the SRS description is related to the SRS title and is clear. A clear SRS description must contain more than three words related to the SRS title. For example, if the SRS title is Hotel Reservation System, the words “hotel” and “reserve” are checked, and the word “system” is excluded as “system” is part of the CSAVL.

- SRS intended users: We checked the presence of intended users of the system in the SRS.
- SRS requirements: Using custom Python scripting, we classified the requirements in each SRS based on the five types of ambiguity generated by the ACM.

Furthermore, subject matter experts comprising six respondents with experience in leading and managing software projects, system analysts, and/or a background in computer science with requirement engineering knowledge were invited to evaluate the SRS. Among the respondents, two had more than 10 years of experience, and four had 2 to 3 years of experience in requirement engineering. The experts were provided with a questionnaire containing 100 SRS documents, and the task was to label fault-prone SRS based on knowledge and understanding. A brief description of fault-prone and clean SRS criteria was also provided to aid the experts in the evaluation. The experts were given 20 days to provide feedback, allowing them to review it at their convenience (Figure 5).

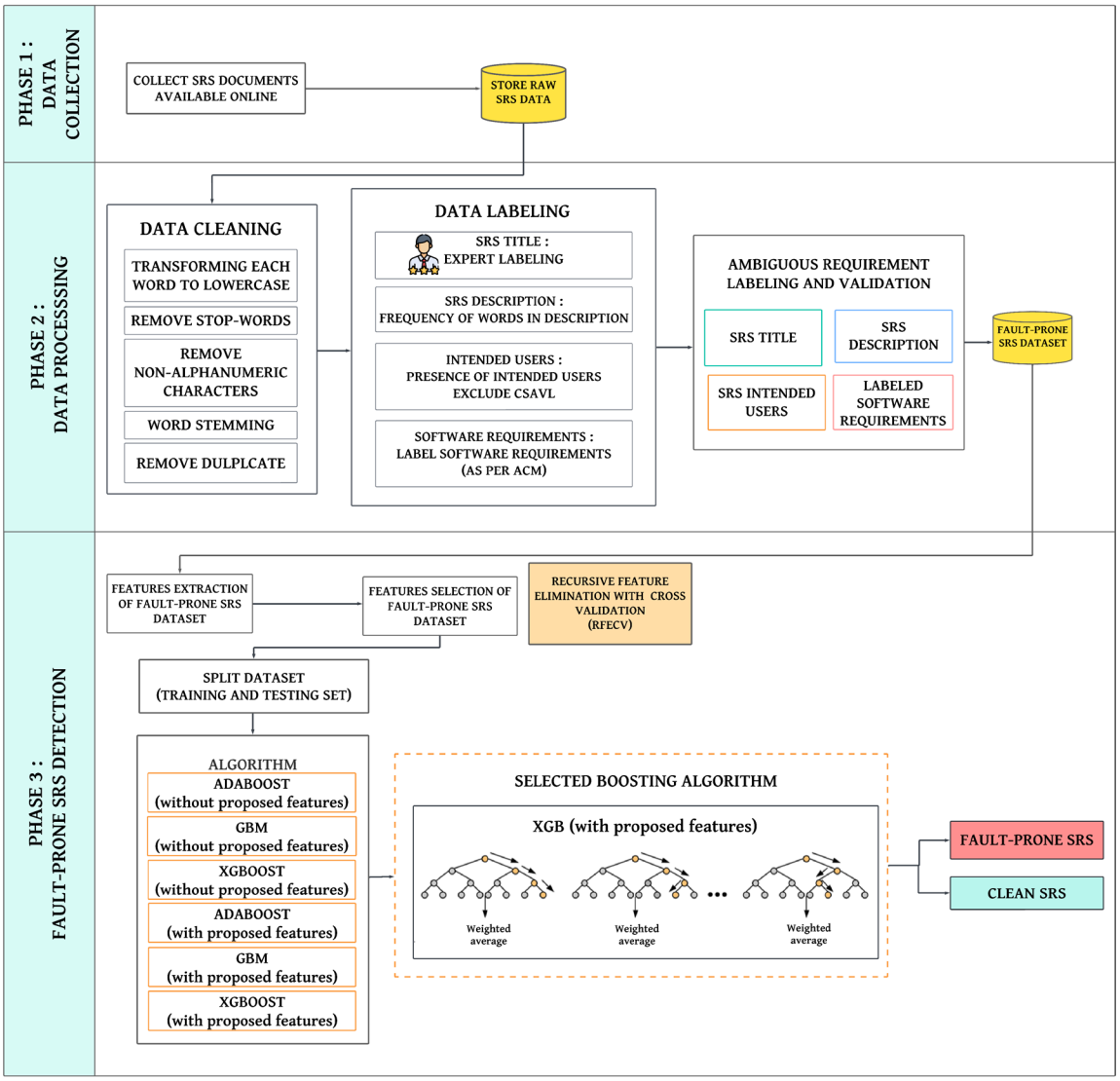


Figure 5. Fault-prone Software Requirements Specification Detection Model.

4.2.3. Phase 3: Fault-Prone Software Requirements Specification Detection

The development of FPDM included statistical feature extraction from the fault-prone software requirements specification dataset. Table 8 shows the list of features that were extracted for each SRS. To prevent overfitting and maintain accuracy in the detection model,

we restricted the number of features to 25 because the number of features is important in building an accurate detection model; too many features might lead to overfitting and increase the complexity of the detection model [22], ultimately reducing the model's accuracy. Reducing the number of features is also important to improve interpretability [23]. The statistical features employed in our model are in the format of double-precision floating point.

Table 8. Extracted statistical-based emotion features based on the ambiguous requirements.

Label	Features	Category	Ref.
F1	Clear title for each SRS	Boolean	Proposed
F2	Frequency of title word in description for each SRS	Numerical	Proposed
F3	Clear description for each SRS	Boolean	Proposed
F4	Description word count for each SRS	Numerical	Proposed
F5	Presence of SRS intended users for each SRS	Boolean	Proposed
F6	Total of software requirements for each SRS	Numerical	[10]
F7	Number of clear software requirements for each SRS	Numerical	Proposed
F8	Number of ambiguous software requirements for each SRS	Numerical	Proposed
F9	Number of lexical ambiguity requirements for each SRS	Numerical	Proposed
F10	Number of syntactic ambiguity requirements for each SRS	Numerical	Proposed
F11	Number of semantic ambiguity requirements for each SRS	Numerical	Proposed
F12	Number of syntax ambiguity requirements for each SRS	Numerical	Proposed
F13	Number of pragmatic ambiguity requirements for each SRS	Numerical	Proposed
F14	Percentage of lexical ambiguity requirements for each SRS	Numerical	[5,7,18]
F15	Percentage of syntactic ambiguity requirements for each SRS	Numerical	[5–7,18]
F16	Percentage of semantic ambiguity requirements for each SRS	Numerical	[7]
F17	Percentage of syntax ambiguity requirements for each SRS	Numerical	[5,6,18]
F18	Percentage of pragmatic ambiguity requirements for each SRS	Numerical	[7,18]
F19	Percentage of clear software requirements for each SRS	Numerical	Proposed
F20	Probability value of lexical ambiguity software requirements for each SRS	Numerical	Proposed
F21	Probability value of syntactic ambiguity software requirements for each SRS	Numerical	Proposed
F22	Probability value of semantic ambiguity software requirements for each SRS	Numerical	Proposed
F23	Probability value of syntax ambiguity software requirements for each SRS	Numerical	Proposed
F24	Probability value of pragmatic ambiguity software requirements for each SRS	Numerical	Proposed
F25	Probability value of clear software requirements for each SRS	Numerical	Proposed

We grouped the set of 25 features that evaluate the quality of software requirements specifications (SRS) into five categories. The first set of features (F1–F4) evaluates the clarity and completeness of the SRS title and description. F1 is a Boolean feature that indicates whether the SRS title is clear and understandable. F2 calculates the frequency of title words in the SRS description. F3 is a Boolean feature that evaluates whether the SRS title contains three or more title words. F4 calculates the total word count of the SRS description. The second set of features (F5–F7) evaluates the identification and completeness of intended users and requirements. F5 is a Boolean feature that indicates whether the intended users of the system are identified in the SRS. F6 calculates the total number of software requirements for each SRS, while F7 calculates the number of clear requirements for each SRS. The third set of features (F8–F13) evaluates the types and frequency of ambiguity in the SRS requirements. F8 calculates the total number of ambiguous requirements for each SRS. F9–F13 are features that calculate the number of each type of ambiguity for each SRS. Specifically, lexical ambiguity (F9), syntactic ambiguity (F10), semantic ambiguity (F11), syntax ambiguity (F12), and pragmatic ambiguity (F13).

The fourth set of features (F14–F18) calculates the percentage of each type of ambiguous requirement for each SRS. Specifically, the percentage of lexical ambiguity (F14), syntactic ambiguity (F15), semantic ambiguity (F16), syntax ambiguity (F17), and pragmatic ambiguity (F18). F19 calculates the percentage of clear requirements for each SRS. Finally, the fifth set of features (F20–F24) calculates the probability value of each type of ambiguity requirement for each SRS. F25 calculates the probability value for clear requirements for each SRS. These features are calculated using Equations (1) and (2), which, respectively, show the formulas to determine the percentage and probability value of each ambiguity class for each SRS. Overall, these features provide a comprehensive evaluation of the quality of SRS, which is crucial for ensuring the success of software development projects. Equation (1) shows the formula to determine the percentage of each type of ambiguity for each SRS, as shown in Equation (1).

$$Percentage\ of\ lexical\ ambiguity = \frac{Total\ number\ of\ lexical\ ambiguity}{Total\ number\ of\ software\ requirements\ for\ each\ SRS} \times 100\% \quad (1)$$

Equation (2) shows the formula to calculate the probability value of each type of ambiguity for each SRS as shown in Equation (2).

$$Probability\ value\ of\ lexical\ ambiguity = \frac{Total\ number\ of\ lexical\ ambiguity}{Total\ number\ of\ software\ requirements\ for\ each\ SRS} \quad (2)$$

In this study, we utilized the recursive feature elimination with the cross validation (RFECV) method to perform feature selection using a feature importance approach. The variable importance function is employed to calculate the feature importance score, which enables the ranking of features based on the significance in decision making. Feature selection is an important process to reduce computational costs and enhance model performance by minimizing the number of input variables. To this end, Hazim et al. [24] suggest several statistical-based features for opinion spam detection, among which two are selected for our study. Similarly, Seri et al. [25] propose 20 statistical-based features, but only seven features are chosen to fit the environment of our study. Specifically, out of the initial 25 features extracted from the dataset, only 12 were selected for the study. Figures 6 and 7 below illustrates the feature importance and correlation matrix.

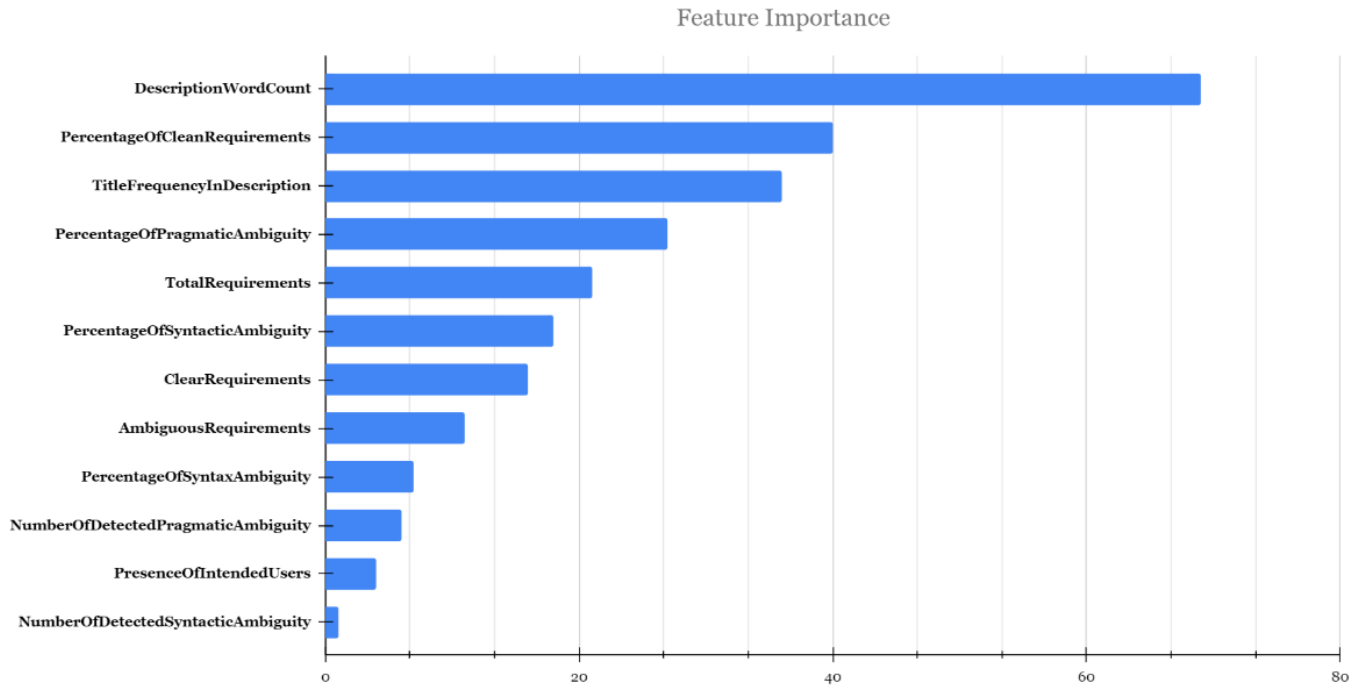


Figure 6. Feature Importance.

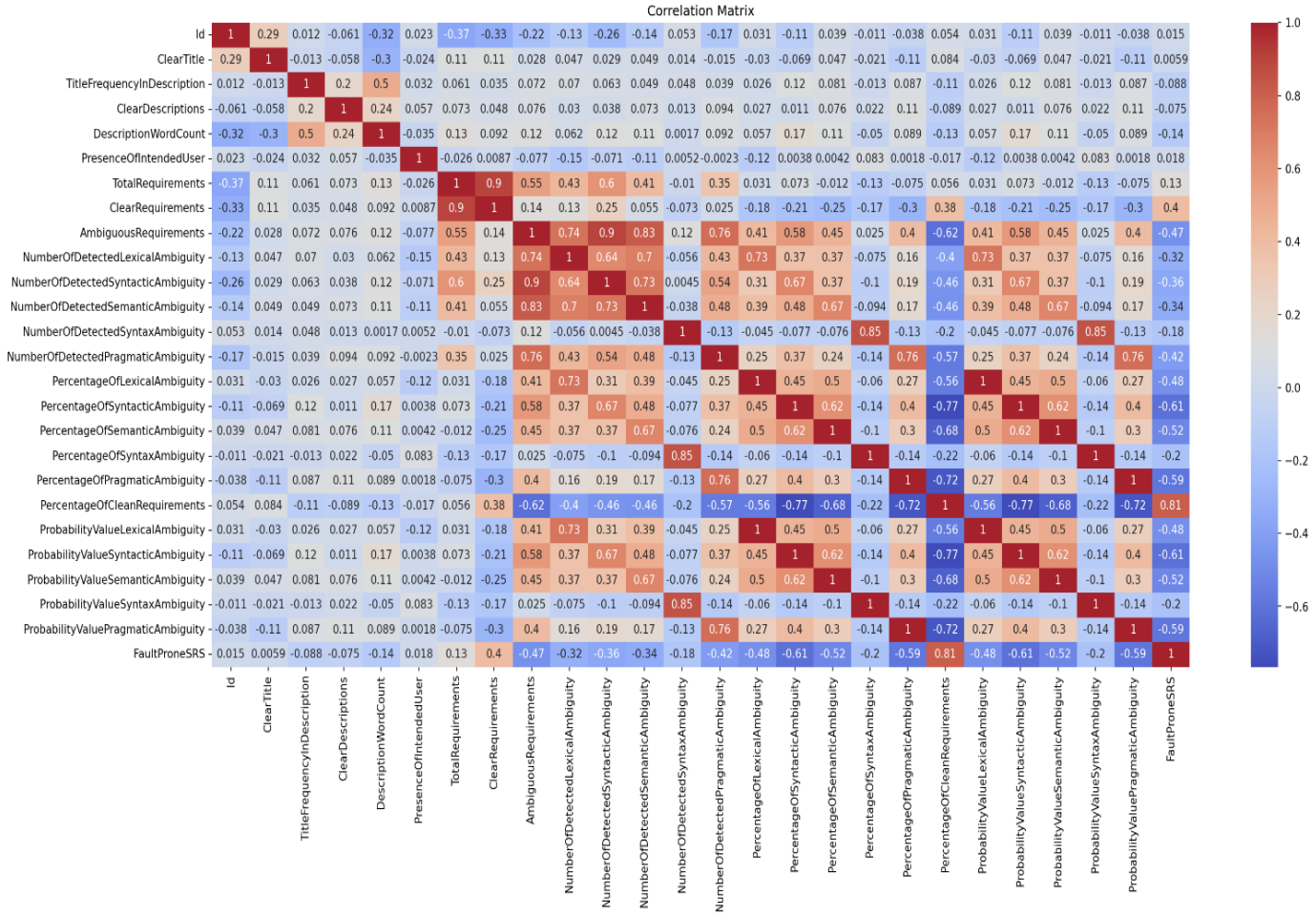


Figure 7. Correlation Matrix.

After implementing AdaBoost, GBM, and XGBoost as the classifiers for the FPDM, we conducted a comparative analysis of the performance. Figure 8 shows the accuracy of each algorithm and it was found that the XGBoost with the proposed features achieved the highest accuracy compared to the others. Table 9 shows the results of the evaluation, including the recall, precision, and f-measure of each model. In this study, the performance of a boosting algorithm was evaluated with and without the proposed features for detecting fault-prone SRS. The evaluation aimed to determine the best model for identifying fault-prone SRS. The FPDM achieved a recall score of 1.000, using XGBoost with proposed features. The recall score implied that XGBoost worked well in detecting the fault-prone SRS by achieving the highest positive rate. In terms of the F-measure score, XGBoost with the proposed features projected an evaluation score of 0.9851, which indicated the ability of the model to balance the positivity rate and false-positive rate. For the precision score, XGBoost with selected existing and proposed features achieved the highest score of 0.9706. Based on these results, the XGBoost model with the proposed features is recommended for identifying fault-prone SRS.

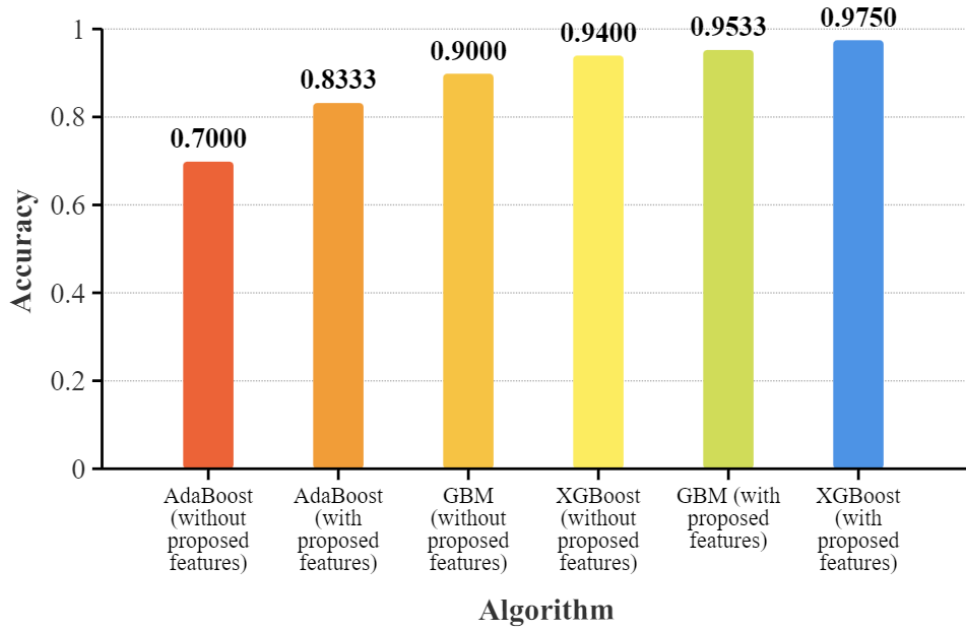


Figure 8. Accuracy of algorithms on the creation of FPDM.

Table 9. Training results for each algorithm in FPDM.

Algorithm	Precision	Recall	F-Measure
AdaBoost (without proposed features)	0.7000	1.000	0.8235
GBM (without proposed features)	0.8700	0.8900	0.8300
XGBoost (without proposed features)	0.9300	0.9100	0.9200
AdaBoost (with proposed features)	0.6667	1.000	0.8000
GBM (with proposed features)	0.9693	0.9752	0.9714
XGBoost (with proposed features)	**0.9706**	**1.000**	**0.9851**

Based on the experimental results presented in Table 10, we presented the comparison of different boosting algorithms in terms of the performance in detecting fault-prone or clean software requirements specifications (SRS) both with and without the proposed features. XGBoost with the proposed features demonstrated the highest accuracy out of all algorithms. The XGBoost algorithm with the proposed features achieved a precision of 0.9800 for fault-prone SRS and 1.000 for clean SRS, recall of 0.1000 for fault-prone SRS and 0.9700 clean SRS, and F-measure of 0.9500 for fault-prone SRS and 0.9900 for clean SRS, which are higher than the corresponding values obtained by other algorithms. The use of the proposed features led to an overall improvement in accuracy for all algorithms. These

findings suggest that XGBoost is the best boosting algorithm for detecting fault-prone SRS, especially when combined with the proposed features.

Table 10. Performance for detecting fault-prone SRS.

Algorithm	SRS	Precision	Recall	F-Measure
AdaBoost (without proposed features)	Fault-prone	0.9200	0.9700	0.9400
	Clean	0.9200	0.8000	0.8600
GBM (without proposed features)	Fault-prone	0.8000	0.8600	0.8300
	Clean	0.9400	0.9200	0.9300
XGBoost (without proposed features)	Fault-prone	0.9500	0.9700	**0.9600**
	Clean	0.9200	0.8500	0.8800
AdaBoost (with proposed features)	Fault-prone	0.9100	0.8300	0.8700
	Clean	0.9500	0.9700	0.9600
GBM (with proposed features)	Fault-prone	**1.0000**	0.7500	0.8600
	Clean	0.8900	**1.0000**	0.9400
XGBoost (with proposed features)	Fault-prone	0.9800	**1.0000**	0.9500
	Clean	**1.000**	0.9700	**0.9900**

5. Case Study: Detecting Fault-Prone Software Requirements Specification for Edge/Cloud Application

5.1. The Evolution of Edge/Cloud Computing

Cloud computing has evolved significantly since the concept emerged in the 1990s. In the 2000s, the use of cloud computing continued to grow with the introduction of new services, such as IaaS and PaaS. In the 2010s, edge computing gained traction with the rise of IoT devices and the need for faster data processing at the edge [26]. Fog computing and open-source platforms for edge computing also emerged during this time. Hybrid cloud solutions and containerization with Kubernetes became key technologies for managing cloud applications in the late 2010s. In 2020, the COVID-19 pandemic led to a surge in demand for cloud services, and serverless computing became more widely adopted [27]. In 2021, AI and ML technologies in cloud services continued to grow, and edge computing saw increased adoption. In 2022, cloud providers focused on making their services more accessible to smaller businesses and individuals while also increasing cloud-based security solutions. Finally, in 2023, quantum computing technologies and blockchain technology are expected to have an impact on cloud computing, with cloud providers offering quantum computing services and blockchain-as-a-service solutions. Figure 9 illustrates the evolution of edge/cloud computing.

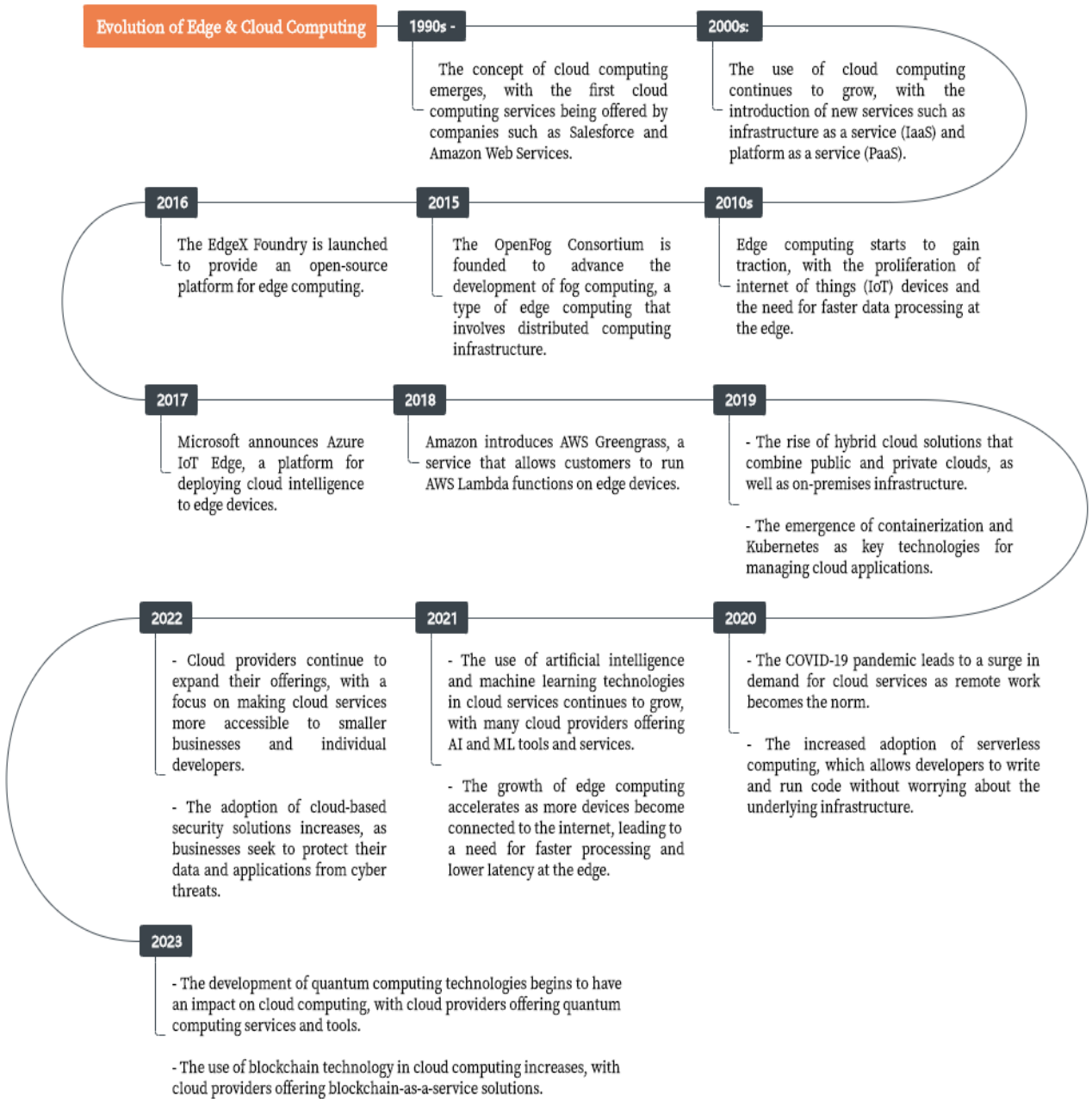


Figure 9. Evolution of Edge and Cloud Computing.

5.2. Edge/Cloud Application Software Requirements Specification

Cloud and edge computing are relatively new paradigms for software development, and, as such, the methods for developing software in these environments are still evolving. The challenges of developing software for cloud and edge computing are that the architecture is often complex and distributed, and there are rapid changes in requirements due to the dynamic nature of these environments, resulting in the difficulty of creating a comprehensive and accurate SRS document. It is a fact that cloud environments are stochastic and dynamic, so it is complex to manage cloud requirements in a systematic and repeatable way, especially when requirements rapidly change in a non-predictive manner [28]. The software requirements specification remains a vital document for cloud computing application development because it specifies the requirements of the cloud computing application, which are essential for the application to function optimally in the cloud environment.

5.3. Analysis of Fault-Prone Edge/Cloud Application Software Requirements Specification

In this case study, we collected 30 SRS documents from various sources, and analyzed them to determine whether the SRS is fault-prone or a clean SRS (Table 11). The SRS documents were related to reservation, management, healthcare, booking, education, and other industries, all of which are based on edge/cloud applications. These documents included the SRS title, description, intended users, and both functional and non-functional requirements. To ensure the quality of these SRS documents, we used a fault-prone software requirements specification model to identify any potential ambiguities in the requirements and identify fault-prone SRS. Overall, this approach allowed us to gain valuable insights into the unique challenges of designing and developing edge/cloud applications across a wide range of industries. By carefully analyzing the SRS, we were able to identify key patterns and trends that inform future development efforts and drive innovation in this rapidly evolving field (Figure 9).

Table 11. Edge/Cloud Applications Project.

Established Year	SRS Project Title	Project Introduction
2017	Istio—Connect Secure Control and Observe Services	Connect Secure Control and Observe Services: Istio is an open-source service mesh platform that provides a way to connect, secure, control, and observe microservices. It provides a powerful set of tools for managing traffic, enforcing policies, and monitoring performance, with built-in support for service-level agreements (SLAs) and other features.
2016	OpenFaaS—Serverless Functions Made Simple	Serverless Functions Made Simple: OpenFaaS is an open-source serverless framework that allows developers to deploy their code as functions to the cloud or on-premises infrastructure. It provides a simple and scalable way to run functions in any language or runtime, with built-in support for Docker containers.
2016	CareKit	CareKit is an open-source framework for developing health and wellness applications for iOS.
2015	MedStack	MedStack is a cloud-based platform for developing and deploying healthcare applications. It provides a secure and compliant environment for developers to build and test their applications, with built-in support for HIPAA and other regulatory requirements.
2015	User Profiling in social media	User profiling in social media refers to the process of analyzing user data from social media platforms to gain insights into user behavior, preferences, and interests. It can be used for targeted advertising, personalized recommendations, and other applications.
2014	Terraform—Infrastructure as Code	Terraform is an open-source tool for building, changing, and versioning infrastructure safely and efficiently. It uses a declarative configuration language to describe infrastructure as code, allowing users to automate the provisioning and management of cloud resources.
2014	CloudMedX	CloudMedX is a cloud-based platform for healthcare data analytics. It uses machine learning and natural language processing to analyze clinical data and generate insights for healthcare providers and payers.
2014	Kubernetes—Automated Container Management	Kubernetes is an open-source platform for automated container management. It provides a powerful set of tools for deploying, scaling, and managing containerized applications, with built-in support for load balancing, service discovery, and other features.
2013	Docker	Docker is an open-source platform for developing, shipping, and running applications in containers. It provides a lightweight and portable way to package and deploy applications, allowing developers to build once and run anywhere.
2013	Roomzilla	Roomzilla is a cloud-based platform for managing conference rooms and other shared spaces. It provides a user-friendly interface for scheduling and availability management.
2013	Appointlet	Appointlet is a cloud-based platform for scheduling appointments and meetings. It provides a customizable booking page and integrations with popular calendar tools.
2012	Prometheus—Monitoring and Alerting	Prometheus is an open-source monitoring and alerting system that collects metrics from different sources, stores them in a time-series database, and provides a powerful query language for analyzing and visualizing them. It also has built-in support for alerting and notifications.
2012	Reservio	Reservio is a cloud-based platform for managing appointments and bookings for businesses of all sizes. It provides a user-friendly interface for scheduling, payment processing, and customer management.

Table 11. *Cont.*

Established Year	SRS Project Title	Project Introduction
2012	RoomKey	A cloud-based hotel property management system (PMS) that allows hoteliers to manage their properties and reservations from a centralized platform. The system includes features such as reservation management, online booking, housekeeping, front desk management, payment processing, and reporting.
2011	Cloud Foundry	Cloud Foundry is an open-source platform for building, deploying, and managing cloud-native applications. It provides a scalable and resilient environment for running applications, with built-in support for continuous integration and delivery (CI/CD) pipelines.
2011	BookingSync	BookingSync is a cloud-based platform for managing vacation rentals, holiday homes, and other short-term rentals.
2011	ClassDojo	ClassDojo is a cloud-based platform for communication and collaboration between teachers, students, and parents. It provides a suite of tools for managing classroom activities, sharing assignments, and providing feedback
2010	OpenStack	OpenStack is an open-source cloud computing platform that provides a set of tools for building and managing private and public clouds. It provides a scalable and flexible infrastructure for running virtual machines, containers, and other cloud-native applications.
2010	HotelTonight	HotelTonight is a cloud-based platform for last-minute hotel bookings. It provides a user-friendly interface for searching and booking hotels, with discounts and special offers.
2009	BookingsPlus	BookingsPlus is a cloud-based platform for managing bookings and reservations for events, facilities, and other resources. It provides a user-friendly interface for booking and payment processing, with built-in support for scheduling and availability management.
2009	Schoology	Schoology is a cloud-based platform for K-12 and higher education institutions. It provides a suite of tools for course management, student engagement, and assessment.
2008	Bookeo	Bookeo is a cloud-based platform for managing bookings and reservations for tours, classes, and other activities. It provides a user-friendly interface for scheduling, payment processing, and customer management.
2008	Cloud Based Hotel Management System	A cloud-based hotel management system is a software platform for managing hotel operations, such as reservations, bookings, payments, and customer service. It provides real-time visibility into hotel activities, with built-in analytics and reporting. It can be used by hotels of all sizes and types.
2007	Cloud based File Sharing System	A cloud-based file sharing system is a software platform for storing and sharing files in the cloud. It provides secure and convenient access to files from any device, with built-in collaboration and version control features. It can be used by individuals, teams, and organizations of all sizes.
2006	Amazon EC2	Amazon Elastic Compute Cloud (EC2) is a web service that provides scalable computing capacity in the cloud. It allows users to launch and manage virtual machines, called instances, on Amazon's infrastructure, providing flexibility and cost savings for a variety of use cases.
2006	Appointy	Appointy is a cloud-based platform for managing appointments and bookings for businesses of all sizes. It provides a user-friendly interface for scheduling, payment processing, and customer management.
2004	OpenMRS	OpenMRS is an open-source electronic medical record system that provides a way to manage patient data in healthcare settings. It is designed to be flexible and customizable, allowing healthcare providers to adapt it to their specific needs.
2002	Cloud based Inventory Management System	A cloud-based inventory management system is a software platform for managing inventory and supply chain operations. It provides real-time visibility into inventory levels, orders, and shipments, with built-in analytics and reporting. It can be used by businesses of all sizes and industries.
1999	Cloud Based Library Management System	A cloud-based library management system is a software platform for managing library operations, such as cataloging, circulation, and patron management. It provides a user-friendly interface for searching and checking out books, with built-in analytics and reporting. It can be used by libraries of all sizes and types.
1997	Athena Health	Athena Health is a cloud-based platform for electronic health records, revenue cycle management, and practice management.

In this study, we utilized two models to evaluate the quality of the software requirements specifications (SRS) for edge/cloud applications: the ambiguity classification model (ACM) and the fault-prone software requirements specification detection model (FPDM). The ACM model was used to classify any potential ambiguities in the software requirements for each SRS, which leads to errors or misinterpretations during the development process. The ACM model flags any ambiguous requirements in the SRS, such as lexical, syntactic, semantic, syntax and pragmatic ambiguities. After applying the ACM model to the SRS dataset, the fault-prone software requirements specification detection model (FPDM) was then used to evaluate the fault-proneness of the SRS documents. The FPDM evaluates the SRS documents based on the title, descriptions, and presence of intended users of the SRS. Additionally, the result based on the ACM for detected ambiguous software requirements was included. This approach allows us to ensure that the SRS documents are of high quality and free of potential faults or ambiguities, ultimately leading to a more successful and efficient development process.

Based on the results of our analysis using the fault-prone software requirements specification model, we determined that out of the 30 SRS documents collected for this study, 20 were classified as fault-prone due to the presence of ambiguities, which could potentially result in errors or misinterpretations during the development process. Conversely, our analysis using the FPDM found that 10 of the 30 SRS documents had a low number of detected ambiguities. These documents had clear titles and descriptions, included information on intended users, and had well-defined functional and non-functional requirements. Based on this, we consider these 10 documents to be relatively free from any major ambiguities or faults. This assessment of the quality of the SRS documents revealed a significant degree of variation, with some documents being clearly and concisely written, while others contained inconsistencies that could impede the development process.

A panel of subject matter experts was assembled for the evaluation of the SRS documents. The panel consisted of six respondents who are experienced in leading and managing software projects, or system analysts with a computer science background and expertise in requirement engineering. Two of the experts had over 10 years of experience, while the remaining four had two to three years of experience in requirement engineering. The experts were presented with a questionnaire that included 30 SRS documents with edge/cloud applications and were tasked with assessing the clarity of the title, adequacy of the system description, suitability of the intended users, and identifying fault-prone SRS based on knowledge and expertise. A brief description of fault-prone and clean SRS criteria was also provided to aid the experts in evaluation. We created a "Labeling tracker" to help the experts to keep track of their progress on labeling the SRS. Additionally, the experts were given 14 days to provide feedback, allowing them to review it at their convenience.

Based on the FPDM and expert evaluation, the classification of the 30 SRS documents was determined (Table 12). However, only two SRS documents did not match. Specifically, while the FPDM detected one SRS document as "fault-prone", the same document was labeled as "clean" by the human experts. Hence, for another document, the FPDM detected one SRS document as "clean", the same document was labeled as "fault-prone" by the human experts. This discrepancy suggests that the FPDM may have a higher sensitivity to identifying potential ambiguities in the requirements compared to human experts. Considering multiple approaches is essential to assess SRS quality to ensure a more accurate and reliable evaluation. Hence, Figure 10 illustrates the comparison of fault-prone and clean SRS based on the year they were established.

Table 12. Comparison of SRS Classification.

Title	SRS Classification		Equal
	Expert Evaluation	Fault-Prone SRS Detection Model	
OpenFaaS—Serverless Functions Made Simple	Clean	Clean	Yes
Terraform—Infrastructure as Code	Clean	Clean	Yes
Prometheus—Monitoring and Alerting	Clean	Clean	Yes
Docker	Clean	Clean	Yes
Amazon EC2	Clean	Clean	Yes
MedStack	Clean	Clean	Yes
User Profiling in social media	Clean	Clean	Yes
CloudMedX	Clean	Clean	Yes
BookingsPlus	Clean	Clean	Yes
Kubernetes—Automated Container Management	Fault-prone	Fault-prone	Yes
Istio—Connect Secure Control and Observe Services	Fault-prone	Fault-prone	Yes
Cloud Foundry	Fault-prone	Fault-prone	Yes
OpenStack	Fault-prone	Fault-prone	Yes
OpenMRS	Fault-prone	Fault-prone	Yes
CareKit	Fault-prone	Fault-prone	Yes
Athena Health	Fault-prone	Fault-prone	Yes
BookingSync	Fault-prone	Fault-prone	Yes
Roomzilla	Fault-prone	Fault-prone	Yes
Appointy	Fault-prone	Fault-prone	Yes
Appointlet	Fault-prone	Fault-prone	Yes
Bookeo	Fault-prone	Fault-prone	Yes
Schoology	Fault-prone	Fault-prone	Yes
ClassDojo	Fault-prone	Fault-prone	Yes
HotelTonight	Fault-prone	Fault-prone	Yes
Cloud based Inventory Management System	Fault-prone	Fault-prone	Yes
Cloud based File Sharing System	Fault-prone	Fault-prone	Yes
Cloud Based Hotel Management System	Fault-prone	Fault-prone	Yes
Cloud Based Library Management System	Fault-prone	Fault-prone	Yes
Reservio	Clean	Fault-prone	No
RoomKey	Fault-prone	Clean	No

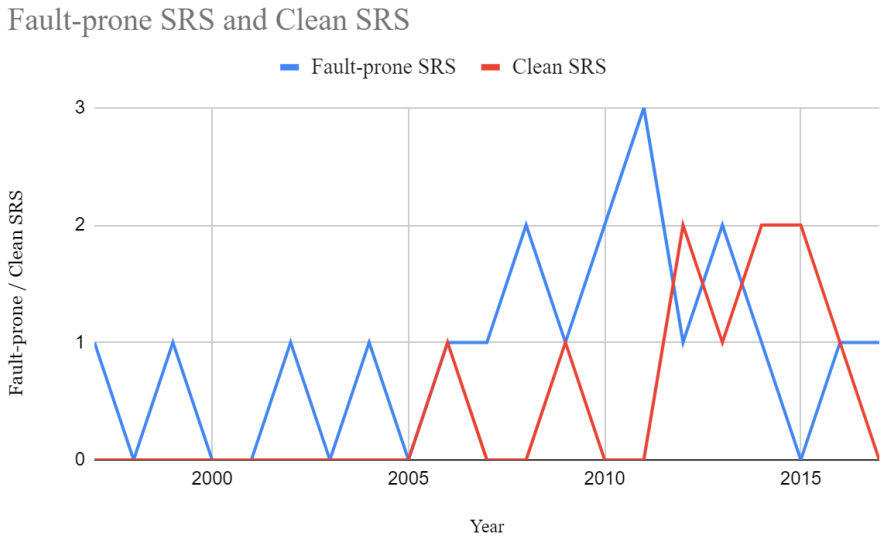


Figure 10. Comparison of fault-prone and clean SRS based on year established.

To achieve this, we utilized three formulas: the linear mapping function for software requirements, ambiguity density index (ADI), and the fault-prone SRS score.

1. Linear Mapping Function for Software Requirements: A mathematical formula used to calculate a score based on a given ratio of clear requirements to ambiguous requirements. The formula uses the concept of exponential decay, where the score decreases as the ratio of clear to ambiguous requirements decreases. The score ranges from 0 to 10, with 10 being the highest score and indicating a high ratio of clear requirements to ambiguous requirements. If the score is high, it means that there are more clear requirements compared to ambiguous requirements. A higher score indicates more clarity and less ambiguity in the requirements.
2. Ambiguity Density Index (ADI): ADI formula quantifies the level of ambiguity in the SRS by measuring the ratio of the number of occurrences of ambiguous words in each SRS to the total number of words in the SRS. A lower ADI value indicates a lower level of ambiguity in the SRS, meaning that the specification is clearer and easier to understand. However, a high ADI value indicates a high level of ambiguity, and areas of the SRS may require further clarification or refinement to ensure a clear and unambiguous specification.
3. Fault-prone SRS Score: The fault-prone SRS Score considers various factors such as the clarity of the title and description, the presence of intended users, the ratio of clear requirements to ambiguous requirements and the usage of ambiguous words to assign a score that reflects the overall fault-proneness of the SRS. These factors are assigned a weight based on the perceived importance in the development of a high-quality SRS.

To clarify, while a higher ADI value generally indicates a higher level of ambiguity it does not necessarily mean that the SRS tends to be fault-prone. Other factors, such as the clarity of the title and description and the presence of intended users, are also capable of impacting the fault-prone score. The fault-prone SRS score is typically used to prioritize testing and quality assurance efforts with SRSs that have a higher score, indicating a lower likelihood of defects or failures in the resulting software product. Figure 11 illustrates the comparison of fault-prone SRS score and ADI for each SRS.

By utilizing these three formulas, we were able to produce a fault-prone severity scale, which serves as a reliable measure of the level of ambiguity and potential for errors in

the SRS for our case study. Equation (3) shows the formula linear mapping function for software requirements (LMF).

$$Linear\ Mapping\ Function\ for\ Software\ Requirements\ (LMF) = 10 * \left(1 - e^{(-x)}\right) \quad (3)$$

$$where\ x = \frac{Clear\ requirements}{Ambiguous\ requirements}$$

Equation (4) shows the formula ambiguity density index (ADI).

$$Ambiguity\ Density\ Index\ (ADI) = \frac{Number\ of\ occurrences\ of\ ambiguous\ words\ in\ each\ SRS}{Total\ number\ of\ words\ in\ each\ SRS} \quad (4)$$

Equation (5) shows the formula fault-prone SRS score as shown in Equation (5).

$$Fault-prone\ SRS\ Score = 10 - [\ (Software\ requirements * w1) + (ADI * w2) + (Title * w3) + (Description * w4\ ) + (Intended\ Users * w5)\ ] \quad (5)$$

Table 13 depicts the distribution of weightage for each factor. A weight of 0.5 was assigned to the software requirements, indicating its importance in determining the clear and ambiguous software requirements in the SRS. A weight of 0.2 was assigned to the ADI, indicating its importance in determining the overall level of ambiguity in the SRS. The clarity of the title and description was assigned a weight of 0.1, reflecting the importance of conveying a clear and concise message to the reader. Finally, the presence of intended users was assigned a weight of 0.1, reflecting the importance of specifying the target audience of the software. With that in mind, the clarity of the title and description was assessed using a score of 0 or 1, where 1 indicates complete clarity and 0 indicates complete ambiguity. Similarly, the presence of intended users in the SRS scored 0 or 1, where 1 indicates the extent to which the document specifies the target audience of the software and 0 indicates none of the intended users are stated in the SRS documents or the intended users stated are not related to the system. We introduced the fault-prone severity scale to determine the degree of ambiguity present in each SRS (Table 14).

Table 13. Weightage distribution.

Factor	Weightage Distribution
Software Requirements	0.5
ADI	0.2
Clear SRS Title	0.1
Clear SRS Description	0.1
Presence of Intended Users	0.1

Table 14. Fault-prone Score SRS.

Title	Ratio Clear to Ambiguous	Software Requirements (LMF)	w1	Total Words	Count Ambiguous Words	ADI	w2	Title	w3	Desc	w4	User	w5	Fault-Prone SRS Score
OpenFaaS—Serverless Functions Made Simple	20:3	9.987316012	0.5	419	85	0.2028639618	0.2	1	0.1	1	0.1	1	0.1	4.665769201
BookingsPlus	21:9	9.027042529	0.5	339	77	0.2271386431	0.2	1	0.1	1	0.1	1	0.1	5.141051007
Prometheus—Monitoring and Alerting	15:6	9.179150014	0.5	298	25	0.08389261745	0.2	1	0.1	0	0.1	1	0.1	5.19364647

Table 14. *Cont.*

Title	Ratio Clear to Ambiguous	Software Requirements (LMF)	w1	Total Words	Count Ambiguous Words	ADI	w2	Title	w3	Desc	w4	User	w5	Fault-Prone SRS Score
Terraform—Infrastructure as Code	16:8	8.646647168	0.5	395	94	0.2379746835	0.2	1	0.1	1	0.1	1	0.1	5.329081479
Amazon Elastic Compute Cloud	16:9	8.313618527	0.5	389	45	0.1156812339	0.2	1	0.1	1	0.1	1	0.1	5.52005449
Docker	11:7	7.919548176	0.5	243	44	0.1810699588	0.2	1	0.1	0	0.1	1	0.1	5.80401192
MedStack	27:18	7.768698399	0.5	670	116	0.1731343284	0.2	1	0.1	0	0.1	1	0.1	5.881023935
User Profiling In Social Media	8:7	6.801809782	0.5	397	32	0.08060453401	0.2	1	0.1	1	0.1	1	0.1	6.282974202
CloudMedX	21:19	6.704410389	0.5	373	58	0.1554959786	0.2	0	0.1	1	0.1	1	0.1	6.41669561
Reservio: Appointment Scheduling Software	19:20	6.132589765	0.5	604	81	0.1341059603	0.2	1	0.1	1	0.1	1	0.1	6.606883925
Kubernetes—Automated Container Management	8:10	5.506710359	0.5	298	138	0.4630872483	0.2	1	0.1	1	0.1	1	0.1	6.854027371
OpenStack	8:10	5.506710359	0.5	438	183	0.4178082192	0.2	1	0.1	0	0.1	1	0.1	6.963083177
Istio—Connect, Secure, Control, and Observe Services	11:16	4.97419775	0.5	454	146	0.3215859031	0.2	1	0.1	1	0.1	1	0.1	7.148583945
Cloud Foundry	7:11	4.705941823	0.5	324	150	0.462962963	0.2	1	0.1	0	0.1	1	0.1	7.354436496
Athena Health	8:15	4.131581992	0.5	337	100	0.296735905	0.2	1	0.1	1	0.1	1	0.1	7.574861823
Cloud-based Hotel Management System	8:16	3.934693403	0.5	556	215	0.3866906475	0.2	1	0.1	1	0.1	1	0.1	7.655315169
Schoology	11:25	3.559635789	0.5	739	187	0.2530446549	0.2	1	0.1	0	0.1	1	0.1	7.969573174
BookingSync: Vacation Rental Software	8:22	3.051088053	0.5	410	136	0.3317073171	0.2	1	0.1	1	0.1	1	0.1	8.10811451
ClassDojo	5:14	3.002275023	0.5	413	118	0.2857142857	0.2	1	0.1	1	0.1	1	0.1	8.141719632
RoomKey	3:10	2.591817793	0.5	393	251	0.6386768448	0.2	1	0.1	1	0.1	1	0.1	8.276355734
HotelTonight	6:21	2.487373841	0.5	315	96	0.3047619048	0.2	1	0.1	1	0.1	1	0.1	8.395360699
Appointy	4:15	2.343269285	0.5	390	101	0.258974359	0.2	1	0.1	1	0.1	1	0.1	8.476570486
Cloud-based Inventory Management System	5:22	2.030792177	0.5	642	257	0.4003115265	0.2	1	0.1	1	0.1	1	0.1	8.604541606
Appointlet	5:21	2.117973089	0.5	508	313	0.6161417323	0.2	1	0.1	0	0.1	1	0.1	8.617785109
CareKit	6:26	2.06260534	0.5	545	130	0.2385321101	0.2	1	0.1	1	0.1	1	0.1	8.620990908
Cloud-based Library Management System	4:18	1.990846357	0.5	470	160	0.3404255319	0.2	1	0.1	1	0.1	1	0.1	8.636491715
Bookeo	4:24	1.538003887	0.5	480	176	0.3666666667	0.2	1	0.1	1	0.1	1	0.1	8.857664723

Table 14. *Cont.*

Title	Ratio Clear to Ambiguous	Software Requirements (LMF)	w1	Total Words	Count Ambiguous Words	ADI	w2	Title	w3	Desc	w4	User	w5	Fault-Prone SRS Score
Roomzilla: Smart Workplace Management System	4:24	1.538003887	0.5	572	167	0.291958042	0.2	1	0.1	1	0.1	1	0.1	8.872606448
Cloud-based File Sharing System	3:19	1.46150218	0.5	544	207	0.3805147059	0.2	1	0.1	1	0.1	1	0.1	8.893145969
OpenMRS	0:19	0	0.5	531	159	0.2994350282	0.2	1	0.1	0	0.1	1	0.1	9.740112994

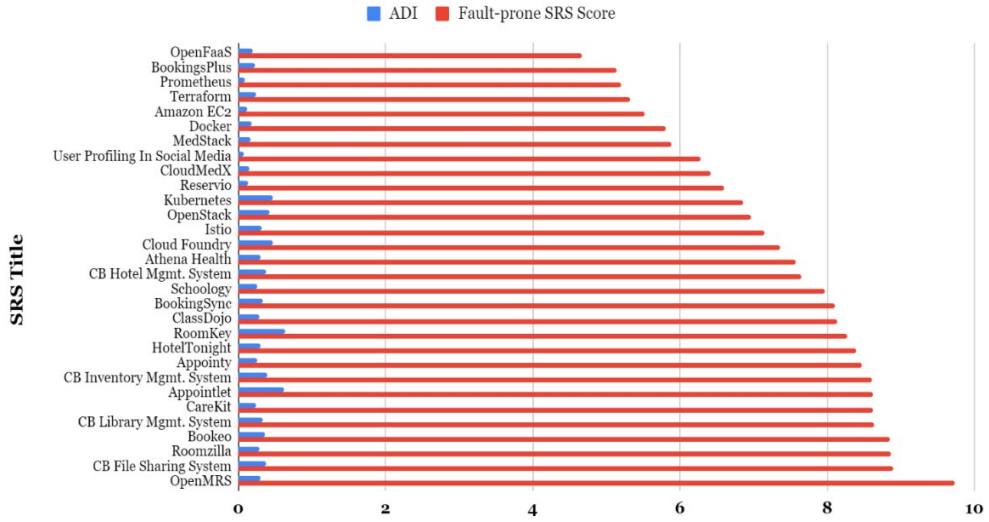


Figure 11. Comparison of ADI and Fault-prone SRS Score for each SRS.

The fault-prone severity scale is a categorization of the level of ambiguity in a software requirements specification (SRS) based on the fault-prone SRS score. The scale is divided into three categories: low ambiguity, moderate ambiguity, and high ambiguity, as illustrated in Figure 12.

Fault-prone Severity Scale	Low	Moderate	High
	0.1–4.9	5.0–6.9	7.0–10.0

Figure 12. Fault-prone Severity Scale Distribution.

The fault-prone severity scale is utilized to assess the level of faults present in individual SRS components. This is typically carried out by evaluating the degree to which a requirement is clear and unambiguous versus how much room there is for interpretation or misinterpretation, clarity of the title and description, and the presence of intended users in the SRS. To determine the level of fault-prone for each SRS, we apply a scale that ranges from low to high ambiguity. The SRS components that are deemed to have a low fault-prone score are clear and unambiguous with no room for interpretation. The SRS components that are deemed to have a moderate fault-prone score are reasonably clear, but there may be some ambiguity or vagueness that requires further clarification. Finally, the SRS that is deemed to have a high fault-prone score is unclear and ambiguous, with a high likelihood of misunderstanding or misinterpretation. One study proposed a method of predicting traffic congestion severity levels based on the analysis of Twitter messages, categorizing them into three levels, i.e., L (low), M (medium), and H (high) [29]. The ratio distribution

of 5:2:3 for the fault-prone severity scale reflects the relative frequency or proportion of the severity levels within the scale. Regarding the distribution ratio of 5:2:3, most of the software requirements and other key components in the SRS are expected to have a low level of ambiguity, while a smaller proportion falls into the moderate and high severity levels. On a serious note, if a fault-prone SRS score is more than 4.9, it means that half of the key components of the SRS, including the title, description, presence of intended user, and software requirements, are having issues.

- **Low level:** For the low severity level, which is defined as scores between 0.1 and 4.9, a ratio of 5 indicates that this level is expected to be the most common and indicates that there is relatively low ambiguity present. This means that most of the requirements and other SRS key components are clear and unambiguous, and no further action is necessary.
- **Moderate Level:** The moderate severity level, which is defined as scores between 5.0 and 6.9, has a ratio of 2. This level indicates that there is some ambiguity present in the software requirements and other SRS key components, which may require further investigation or clarification.
- **High Level:** A high severity level, which is defined as scores between 7.0 and 10.0, has a ratio of 3. This level indicates that there is a significant amount of ambiguity present, which is capable of leading to significant issues if not addressed. Therefore, more attention is needed to resolve the ambiguity at this level compared to the moderate level.

Table 15 and Figure 13 show a list of cloud and edge applications with their respective fault-prone SRS scores and fault-prone severity scales. The fault-prone SRS score ranges from 4.665769201 (low) to 9.740112994 (high). OpenFaaS, with a score of 4.665769201, has the lowest score on the list, while OpenMRS, with a score of 9.740112994, has the highest score. Most of the applications fall within the moderate range, with a score between 5 and 7.999999999. In terms of year, the list shows that the applications range from 1997 to 2017. It is interesting to note that the applications with the highest scores are generally from earlier years, with Athena Health being the oldest on the list. This may be due to the fact that these applications were developed at a time when cloud technology was less mature and less well understood, increasing the potential for errors in the SRS. Some of the technologies with the highest fault-prone SRS scores and high fault-prone severity levels include OpenMRS, the Cloud-based file sharing system, Roomzilla Bookeo, and the Cloud-based library management system. Referring to Figure 13, there are several reasons why later projects in edge/cloud applications with high fault-prone SRS scores are due to possible complexities and rapid development.

Table 15. Fault-prone Severity Scale.

Title	Fault-Prone SRS Score	Fault-Prone Severity Scale
OpenFaaS—Serverless Functions Made Simple	4.665769201	Low
BookingsPlus	5.141051007	Moderate
Prometheus—Monitoring and Alerting	5.19364647	Moderate
Terraform—Infrastructure as Code	5.329081479	Moderate
Amazon Elastic Compute Cloud	5.52005449	Moderate
Docker	5.80401192	Moderate
MedStack	5.881023935	Moderate
User Profiling In Social Media	6.282974202	Moderate
CloudMedX	6.41669561	Moderate
Reservio: Appointment Scheduling Software	6.606883925	Moderate
Kubernetes—Automated Container Management	6.854027371	Moderate
OpenStack	6.963083177	Moderate
Istio—Connect, Secure, Control, and Observe Services	7.148583945	High

Table 15. *Cont.*

Title	Fault-Prone SRS Score	Fault-Prone Severity Scale
Cloud Foundry	7.354436496	High
Athena Health	7.574861823	High
Cloud-based Hotel Management System	7.655315169	High
Schoology	7.969573174	High
BookingSync: Vacation Rental Software	8.10811451	High
ClassDojo	8.141719632	High
RoomKey	8.276355734	High
HotelTonight	8.395360699	High
Appointy	8.476570486	High
Cloud-based Inventory Management System	8.604541606	High
Appointlet	8.617785109	High
CareKit	8.620990908	High
Cloud-based Library Management System	8.636491715	High
Bookeo	8.857664723	High
Roomzilla: Smart Workplace Management System	8.872606448	High
Cloud-based File Sharing System	8.893145969	High
OpenMRS	9.740112994	High

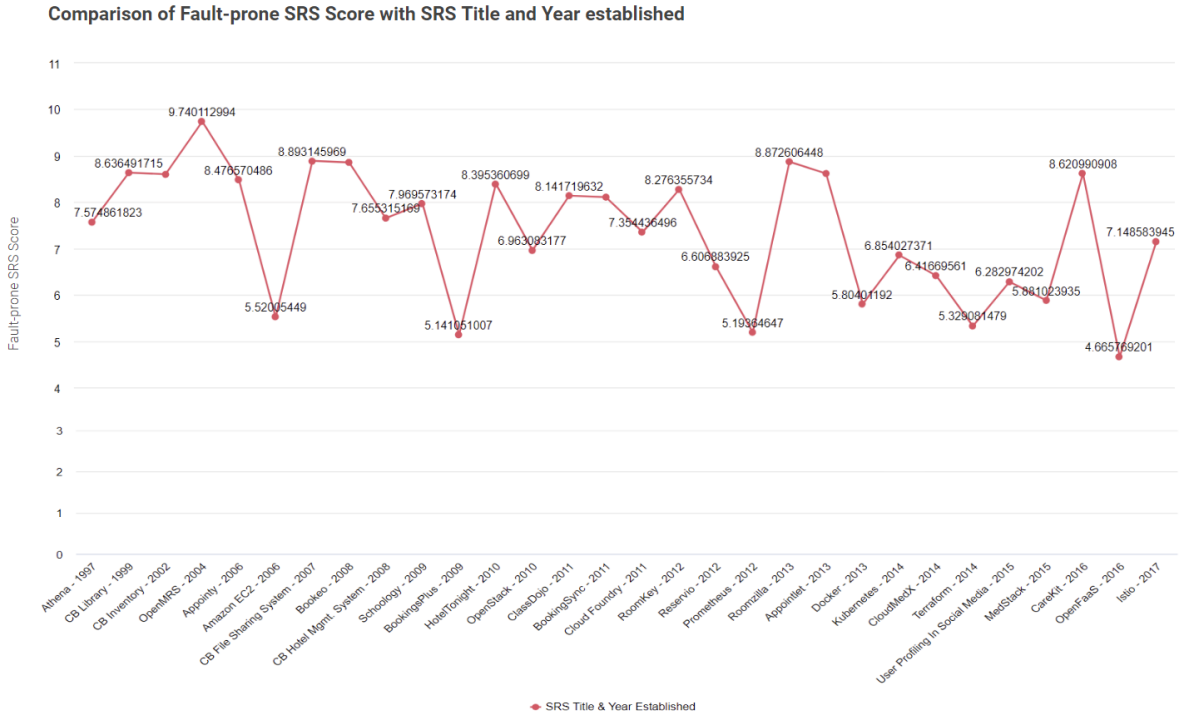


Figure 13. Comparison of Fault-prone SRS Score with SRS title and Year established.

Edge/cloud applications are becoming increasingly complex as they involve distributed systems, multiple devices, and heterogeneous networks, which makes it challenging to define precise and unambiguous requirements. Many edge/cloud applications are developed rapidly to meet fast-changing market demands; hence, the SRS may be hastily written, leading to errors and inaccuracies that can increase the fault-proneness of the software. Cloud computing is very complex to administrate because of the dynamism imposed by the context, stochastic requirements depending on business changes, heterogeneous consumers from different places and jurisdictions, distributed systems, and remote management [28]. Considering the unique requirements of edge computing when developing

software systems, it is important that the software requirements specification (SRS) of the system plays a major role in ensuring that the software system is designed and developed to meet the unique requirements of the edge computing environment. Our analysis indicates that the fault-prone software requirements specification model (FPDM) is a dependable method for detecting fault-prone SRS in edge/cloud application development. The early detection and resolution of these fault-prone areas using the FPDM enhances the reliability and overall quality of the resulting application.

6. Conclusions

This paper presents a fault-prone software requirements specification detection model that aims to detect fault-prone software requirements specifications (SRS). The model comprises two parts: the ambiguity classification model (ACM) and the fault-prone software requirements specification detection model (FPDM). The ACM utilizes different deep learning algorithms. Then, the best performance algorithm is selected to classify each software requirement as either ambiguous or clean. The ACM is capable of classifying and detecting the presence of ambiguous requirements in the software requirements specification (SRS), covering a range of ambiguities, including lexical, syntactic, semantic, syntax, and pragmatic ambiguities. The FPDM then uses the key components of SRS—clarity of title and description, presence of intended users, and classified ambiguous requirements—to detect fault-prone SRS. The ACM achieved an accuracy of 0.9907, while the FPDM achieved an accuracy of 0.9750. This study also explored the use of statistical-based features to improve the performance of the FPDM, with a restriction on the number of features to prevent overfitting and maintain accuracy. Subject matter experts were also invited to evaluate the SRS documents for fault-proneness. The implementation of a boosting model was found to enhance the model's accuracy in detecting fault-prone SRS for edge/cloud applications.

Developing edge/cloud applications is becoming more challenging due to their distributed nature, heterogeneous networks, and fast-changing market demands. This complexity leads to hastily written and ambiguous software requirements, increasing the fault-proneness of the resulting software. Cloud computing's dynamism and distributed nature further exacerbate these challenges. Considering the distinctive software requirements of edge computing, the SRS plays a critical role in ensuring software systems are developed to meet these demands. Utilizing the FPDM, faults in the SRS were detected and resolved early, enhancing the reliability and quality of the application. In our case study, we developed a fault-prone severity scale to assess the level of ambiguity and potential for errors in the software requirements specification (SRS). To achieve this, we used three formulas, namely the linear mapping function for software requirements, the ambiguity density index (ADI), and the fault-prone SRS score. The utilization of these three formulas allowed us to generate a reliable measure of the severity of faults in the SRS.

The continuous advancements in technology and the implementation of boosting algorithms reduced the need for human intervention in the development of predictive models. However, despite the potential of these algorithms to identify and reaffirm the fault-proneness of software requirements specifications (SRS), human intervention is still necessary to interpret the clarity and ambiguity of SRS components. The findings of this study show that boosting algorithms and human expertise improve the accuracy and effectiveness of fault-prone detection models in identifying and distinguishing between ambiguous and clear software requirements and the clarity of other SRS components. Further research is necessary to explore the implementation of boosting algorithms in the development of predictive models to enhance the identification of clean and fault-prone SRS documents.

Author Contributions: F.N.J.M. served as the main author and took a leading role in all aspects of the project, including research, development, and analysis, S.H.A.H. as a co-author significantly contributed by providing numerous ideas and expertise specifically related to fault-prone detection, utilizing boosting techniques consistently throughout the project, H.S. provided valuable support in conducting an extensive literature review, ensuring a solid theoretical foundation for the research,

R.A.R. played a crucial role in gathering and organizing the dataset used for the project's analysis, F.F. made important contributions by offering insightful ideas and suggestions throughout the duration of the project. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by University of Malaya Research Grant (ST014-2022).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Written informed consent has been obtained from the expert(s) to publish this paper.

Data Availability Statement: The dataset supporting the reported results of this research study has been made available for public access. It can be found at the following location: https://www.kaggle.com/datasets/corpus4panwo/fault-prone-srs-dataset.

Acknowledgments: We gratefully thank the Graduate Excellence Programme (GrEP) of Majlis Amanah Rakyat for awarding the scholarship to support the studies and this research is funded by University of Malaya Research Grant (ST014-2022) Corpus Development for Anxiety Disorder Profile Detection Model on Twitter Communication using Fear and Worry Emotion Analytics project.

Conflicts of Interest: The authors declare that there are no conflict of interest regarding the publication of this manuscript.

References

1. Sivarajah, S.; Irani, M.A.; Weerakkody, C.; Akter, R.L.E. Critical analysis of Big Data challenges and analytical methods. *J. Bus. Res.* **2017**, *70*, 263–286. [CrossRef]
2. Mitchel. Leveraging Natural Language Processing in Requirements Analysis. QRA Corp. 16 February 2021. Available online: https://qracorp.com/nlp-requirements-analysis/ (accessed on 14 March 2022).
3. Sabriye, A.O.J.; Zainon, W.M.N.W. A framework for detecting ambiguity in software requirement specification. In Proceedings of the 2017 8th International Conference on Information Technology (ICIT), Amman, Jordan, 17–18 May 2017. [CrossRef]
4. Singh, M.; Walia, G.S. Automating Key Phrase Extraction from Fault Logs to Support Post-Inspection Repair of Software Requirements. In Proceedings of the India Software Engineering Conference, Bhubaneswar, India, 25–27 February 2021. [CrossRef]
5. Nigam, A.; Arya, N.; Nigam, B.; Jain, D. Tool for Automatic Discovery of Ambiguity in Requirements. *Int. J. Comput. Sci. Issues* **2012**, *9*, 350.
6. Sabriye, A.O.J.; Zainon, W.M.N.W. An approach for detecting syntax and syntactic ambiguity in software requirement specification. *J. Theor. Appl. Inf. Technol.* **2018**, *96*, 2275–2284.
7. Rani, A.; Aggarwal, G. Algorithm for Automatic Detection of Ambiguities from Software Requirements. *Int. J. Innov. Technol. Explor. Eng. (IJITEE)* **2019**, *8*, 878–882. [CrossRef]
8. IBajwa, S.; Lee, M.; Bordbar, B. *Resolving Syntactic Ambiguities in Natural Language Specification of Constraints*; Springer eBooks: Berlin/Heidelberg, Germany, 2012; pp. 178–187. [CrossRef]
9. Ferrari, A.; Donati, B.; Gnesi, S. Detecting Domain-Specific Ambiguities: An NLP Approach Based on Wikipedia Crawling and Word Embeddings. In Proceedings of the 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), Lisbon, Portugal, 4–8 September 2017. [CrossRef]
10. Osman, M.H.; Zaharin, M.F. Ambi Detect: An Ambiguous Software Requirements Specification Detection Tool. *Turk. J. Comput. Math. Educ.* **2021**, *12*, 2023–2028. [CrossRef]
11. Kurtanovic, Z.; Maalej, W. Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning. In Proceedings of the IEEE International Conference on Requirements Engineering, Lisbon, Portugal, 4–8 September 2017. [CrossRef]
12. Alshazly, A.A.; Elfatatry, A.; Abougabal, M.S. Detecting defects in software requirements specification. *Alex. Eng. J.* **2014**, *53*, 513–527. [CrossRef]
13. Singh, M. Automated Validation of Requirement Reviews: A Machine Learning Approach. In Proceedings of the IEEE International Conference on Requirements Engineering, Banff, AB, Canada, 20–24 August 2018. [CrossRef]
14. *IEEE Std 830-1998*; IEEE Recommended Practice for Software Requirements Specifications. IEEE: New York, NY, USA, 1998.
15. Ali, S.; Khan, N.A.; Alshayeb, M.; Alghamdi, A. An Empirical Study of the Impact of SRS Quality on the Fault Proneness of Software Systems. In Proceedings of the 9th International Conference on Software Engineering and Service Science, Beijing, China, 23–25 November 2018; pp. 279–283.
16. Aggarwal, A.; Singh, S.; Kaur, N. A Study of Software Requirement Specification. *Int. J. Comput. Appl.* **2015**, *126*, 1–6.
17. Bäumer, F.S.; Geierhos, M. Flexible Ambiguity Resolution and Incompleteness Detection in Requirements Descriptions via an Indicator-Based Configuration of Text Analysis Pipelines. In Proceedings of the Annual Hawaii International Conference on System Sciences, Hilton Waikoloa Village, HI, USA, 3–6 January 2018. [CrossRef]
18. Osama, S.; Aref, M. Detecting and resolving ambiguity approach in requirement specification: Implementation, results and evaluation. *Int. J. Intell. Comput. Inf. Sci.* **2018**, *18*, 27–36. [CrossRef]

19. Pennington, J.; Socher, R.; Manning, C.D. Glove: Global Vectors for Word Representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014. [CrossRef]
20. Mikolov, T.; Chen, K.; Corrado, G.S.; Dean, J. Efficient Estimation of Word Representations in Vector Space. *arXiv* **2013**, arXiv:1301.3781.
21. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [CrossRef]
22. Firdaus, A.; Anuar, N.B.; Razak, M.A.A.; Sangaiah, A.K. Bio-inspired computational paradigm for feature investigation and malware detection: Interactive analytics. *Multimed. Tools Appl.* **2018**, *77*, 17519–17555. [CrossRef]
23. Virgolin, M.; Alderliesten, T.; Bosman, P.A.N. On explaining machine learning models by evolving crucial and compact features. *Swarm Evol. Comput.* **2019**, *53*, 100640. [CrossRef]
24. Hazim, M.; Anuar, N.B.; Razak, M.F.A.; Abdullah, N.A. Detecting Opinion Spams through Supervised Boosting Approach. *PLoS ONE* **2018**, *13*, e0198884.
25. Razak, C.S.A.; Hamid, S.H.A.; Meon, H.; Hema, A.; Subramaniam, P.; Anuar, N.B. Two-step model for emotion detection on twitter users: A Covid-19 case study in Malaysia. *Malays. J. Comput. Sci.* **2021**, *34*, 374–388. [CrossRef]
26. Ranger, S. What Is Cloud Computing? Everything You Need to Know About the Cloud Explained. ZDNET. 25 February 2022. Available online: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/ (accessed on 18 May 2023).
27. Aggarwal, G. How The Pandemic Has Accelerated Cloud Adoption. Forbes. 15 January 2021. Available online: https://www.forbes.com/sites/forbestechcouncil/2021/01/15/how-the-pandemic-has-accelerated-cloud-adoption/?sh=3ca3ba626621 (accessed on 18 May 2023).
28. Zalazar, A.S.; Ballejos, L.C.; Rodriguez, S. Analyzing Requirements Engineering for Cloud Computing. In *Requirements Engineering for Service and Cloud Computing*; Ramachandran, M., Mahmood, Z., Eds.; Springer: Cham, Switzerland, 2017. [CrossRef]
29. Wongcharoen, S.; Senivongse, T. Twitter analysis of road traffic congestion severity estimation. In Proceedings of the 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), Khon Kaen, Thailand, 13–15 July 2016; pp. 1–6. [CrossRef]