Design and Development of Reusable Feature-based Next-Generation Embedded Software

by

Md Al Maruf

A thesis submitted to the

School of Graduate and Postdoctoral Studies in partial

fulfillment of the requirements for the degree of

Doctor of Philosophy (Ph.D.) in Electrical and Computer Engineering

Department of Electrical, Computer and Software Engineering

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

December 2023

Thesis Examination Information

Submitted by: **Md Al Maruf**

Doctor of Philosophy in **Electrical and Computer Engineering**

Thesis Title: Design and Development of Reusable Feature-based Next-Generation Embedded Software

An oral defense of this thesis took place on November 16, 2023 in front of the following examining committee:

Examining Committee:

Research Supervisor: **Dr. Akramul Azim**

Committee Member: **Dr. Qusay H. Mahmoud**

Committee Member: **Dr. Sanaa Alwidian**

University Examiner: **Dr. Mohamed El-Darieby**

External Examiner: **Dr. Andriy Miranskyy**

Chair of Examining Committee: **Dr. Ramiro Liscano**

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

The emergence of next-generation embedded systems, emphasized by their shortened life cycles, necessitates an urgent shift towards agile software design and development. The objective is to achieve timely product delivery while maintaining safety and quality standards. Anticipating that next-generation software will interconnect numerous devices, an efficient architecture supporting advanced functionalities or features becomes essential. Fog and edge computing emerge as promising computing paradigms for next-generation embedded applications. These platforms are particularly pertinent for safety-critical and time-sensitive systems, such as autonomous vehicles. However, integrating these platforms into embedded systems presents challenges in designing and developing software supporting future demands like mobility and machine learning (ML) model training.

This research focuses on identifying the reusable features through static analysis from legacy embedded software to improve code reuse for faster development and create a feature model for understanding features and their requirements. The feature model displays embedded software's integrated variants and constraints details to reduce the feature verification and validation effort. It supports reusability, significantly easing key development phases such as requirement analysis, which is often a major bottleneck in the timely release of embedded software, even with agile methodologies. Further, the study emphasizes designing fog computing architecture that benefits embedded applications like over-the-air (OTA) software updates and improves the performance of large ML model training by efficient model partitioning across edge devices. Our research presents a feature-based embedded software development approach that incorporates the advanced features in the feature model and streamlines the entire development cycle from design to deployment. A Python tool is developed to automatically extract reusable features from publicly available GitHub embedded software projects, showcasing the practical applicability of our research in real-world scenarios.

Keywords— Embedded Software Design and Development, Software Reuse, Feature and Requirements Identification, Feature Model, Fog and Edge Computing, Machine Learning Model Parallelism

Author's Declaration

Md Al Maruf

Statement of Contributions

The research topic and the work described in this thesis have been published to the following events:

a. Feature-based Reusable Embedded Software Design:

1. *Conference:* **Al Maruf, M.**, & Azim, A. "WiP: Automated Features and Requirements Identification for Improving CPS Software Reuse using Topic Modeling" Proceedings of the 14th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS), San Antinio, Texas, USA. (2023)

2. *Journal:* **Al Maruf, M.**, & Azim, A. "Facilitating Reuse of Functions in Embedded Software" IEEE Access 10 : 88595-88605. (2022)

b. Next-Generation Embedded Applications Design:

3. *Conference:* **Al Maruf, M.**, & Azim, A. "Towards Safe Online Machine Learning Model Training and Inference on Edge Networks" IEEE 22nd International Conference on Machine Learning and Applications (ICMLA), Jacksonville, Florida, USA. (2023)

4. *Conference:* **Al Maruf, M.**, & Azim, A. "Optimizing DNNs Model Partitioning for Enhanced Performance on Edge Devices" Proceedings of the 36th Canadian Conference Artificial Intelligence (Canadian AI), Montreal, Canada. (2023)

5. *Book Chapter:* Azim, A., & **Al Maruf, M.** "Cognitive Mobile Computing for Cyber-Physical Systems (CPS)" Towards a Wireless Connected World: Achievements and New Technologies. Springer International Publishing (pp. 203–222). (2022)

6. *Journal:* **Al Maruf, M.**, Singh, A., Azim, A., & Auluck, N. "Faster Fog Computing based Over-the-air Vehicular Updates: A Transfer Learning Approach." IEEE Transactions on Services Computing (TSC), IF: 11.019. (2021).

7. *Conference:* **Al Maruf, M.**, Singh, A., Azim, A., & Auluck, N. "Resource efficient allocation of fog nodes for faster vehicular OTA updates." IEEE International Symposium on Networks, Computers and Communications (ISNCC) (pp. 1-6), Montreal, Canada. (2020).

I hereby certify that I am the main author of this thesis. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

Acknowledgements

I would like to begin by expressing my deepest gratitude to the Almighty Allah for granting me the strength and perseverance to complete this PhD thesis.

I am profoundly grateful to my supervisor, Dr. Akramul Azim, for his invaluable guidance, support, and constant encouragement throughout the research and writing process of this thesis. Dr. Azim's insightful comments, suggestions, and unwavering belief in my abilities have been instrumental in my academic journey. His office door was always open, and he provided the right direction whenever I faced challenges in my research. I am also grateful to the faculty members and staff of Ontario Tech University for their assistance and guidance. Their support, including the generous financial support provided by the university, has played an essential role in my academic success. I would also like to thank all the RTEMSOFT and SIRC-3340 research lab members, who have offered their support, constructive criticism, and thoughts to help improve my work.

Finally, I must extend my deepest and most heartfelt gratitude to my family for their support and encouragement throughout my years of study. To my beloved mother, Mahfuza Begum, father Motalleb Hossain, whose unconditional love and prayer have been my anchor. To my sister Khairun Nesa, brother Naim Ali, and brother-in-law Sulaiman Sumon for their enduring support and belief in me. This achievement is as much theirs as it is mine. I dearly wish my grandparents were here today to see this achievement; their memory remains a constant source of inspiration. To my in-laws and loving wife, Rezwana Mamata, words cannot express my gratitude for your continuous support, love, and understanding throughout this journey. Rezwana, your constant encouragement and faith in me have been a source of strength, helping me overcome my challenges during my Ph.D. Thank you.

I would also like to acknowledge the support of my friends, colleagues, and relatives who have always motivated and encouraged me throughout this journey. Your belief in me and your uplifting words have truly made a difference in my life.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPS	Cyber-Physical Systems
ML	Machine Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
BERT	Bidirectional Encoder Representations from Transformers
ECU	Electronic Control Unit
FCA	Fiat Chrysler Automobiles
ISR	Interrupt Service Routines
OTA	Over-the-Air
FM	Feature Model
SPL	Software Product Line
RSUs	Road Side Units
SVM	Support Vector Machine
RF	Random Forest
TMR	Tripple Modular Redundancy
AST	Abstract Syntax Trees
DAC	Data Abstraction Coupling
IoT	Internet of Things
CBO	Coupling Between Object
AFM	Adaptive Feature Model
FIG	Feature Interaction Graph
LDA	Latent Dirichlet Allocation
TF-IDF	Term Frequency - Inverse Document Frequency

Chapter 1

INTRODUCTION

The demand for next-generation embedded software solutions, especially in Cyber-Physical Systems (CPS), is rapidly growing [1]. Examples of such solutions include automotive electronic control unit (ECU) software, networked systems for medical devices, and industrial control systems. Modern embedded systems are software-intensive, integrating sensors, logic, power systems, operating systems, resource managers, and communication networks to serve various applications. For instance, next-generation embedded software in autonomous vehicles demonstrates the potential for enabling new computing paradigms (e.g., fog or edge computing) to enhance control, computation, and communication technologies. Furthermore, these systems must satisfy numerous constraints to ensure service safety and quality.

As the landscape of embedded software expands and evolves, traditional manufacturers are in a race against time. The competition in the software market is intense, with every entity striving to release products before their rivals. With ever-changing embedded software requirements, manufacturers need to update and produce new software frequently. Any delay or misstep in designing and developing according to the new requirements could cause the software to become obsolete. This competition intensifies with the integration of advanced functionalities into next-generation embedded systems, including machine learning-based applications, Over-the-Air (OTA) software updates, and faster computing services. The requirements of these advanced functionalities in this context refer to the demands for higher performance (e.g., faster computation), automation, security, interconnectivity, and adaptability in embedded software, lead to new challenges. These challenges manifest in the form of implementing constraints related to time, resources, and design. Unlike traditional software, embedded software has its nuances, such as interrupt handling using interrupt service routines (ISR), task scheduling, and language dependence. It has been found that over 80% of all embedded systems are developed using the C procedural and function-oriented language [2]. In many cases, other languages are considered memory or resource-intensive for effective deployment in bare-metal development. Building embedded software from scratch is time-consuming and requires skilled developers who can understand the necessary requirements for implementing software functionalities. Each software feature undergoes extensive testing to verify its functionalities.

Yet, amidst these challenges lies an opportunity: many next-generation embedded software share features with legacy software. Reusing these features could potentially reduce implementation and testing time, which are major bottlenecks in software release. Additionally, software reusability can increase productivity, accelerate development, and decrease operational costs. Generally, software practitioners reuse code to shorten development time when implementing required functionalities. Nevertheless, large-scale code reuse remains scarce in embedded software development due to its complex constraints. Furthermore, much of the publicly available embedded software source code lack proper documentation, making it difficult for developers to understand system features and their requirements for reuse. As a result, developers often hesitate to reuse legacy embedded software code found in public repositories, such as GitHub.

In this dissertation, a *feature is defined as* a reusable distinct unit of functionality that can be encapsulated within a computational unit or a function [3] to satisfy any application-specific task [4], [5]. In a broader perspective, features can be functional (representing a specific behavior or service the system provides), or non-functional (relating to the system's properties or quality, such as performance or security).

In recent years, numerous software engineering techniques, such as programming language constructs and software product line visualization, have employed object-oriented structures for software development. However, embedded software typically executes specialized tasks as functions [6] that frequently interact with other computational units [7]. For example, the adaptive cruise control function of automotive embedded systems is distributed across multiple Electronic Control Units (ECUs) and repeatedly interacts with other functions dispersed throughout numerous ECUs [8]. Identifying these reusable functions is challenging when they are located in different parts of the code without adhering to proper coding practices. Recent software reuse studies [9], [10] recognize these functions as potential features for reuse aligning with the requirements. However, approaches that automatically extract features and requirements from embedded source code are relatively few and far between.

Current techniques for feature and requirements extraction often resort to manual methods, which are time-consuming and prone to errors [11]–[14]. While Software Product Line (SPL) [10] approaches offer promise in developing applications using reusable assets, many of these methods lean towards object-oriented code and neglect legacy C-based embedded software. This trend has led to gaps in understanding the potential of reusable components within source codes and generating feature models tailored for embedded systems. Furthermore, developers face challenges in determining configurations of reusable features from given source code for developing a specific application. However, with the advent of technologies like BERT and topic modeling for static analysis [15], [16], there is potential to revolutionize this process. The Bidirectional Encoder Representations from Transformers (BERT) [17] model assists in discerning semantic distinction in code and associated documentation, while the topic modeling procedure captures the structural essence within the codebase [18]. These technologies could vastly improve feature and requirement extraction from legacy codes, bringing clarity to software's variability and aiding its design and development.

There is untapped potential in reusing legacy software to develop next-generation embedded software, which can lead to reduced development time and increased reliability. While inherent constraints and the lack of legacy code documentation have deterred extensive reuse in the past, publicly available repositories, such as GitHub, hold promise in this domain. Therefore, in this research, we introduce an approach to automatically identify reusable functions, potential features and requirements, vital for the evolution of next-generation embedded software. Building on the foundational use of a function call graph, our method ranks program functions and extracts calling relationships between subroutines, encapsulating function dependencies and constraints [19], [20]. Our strategy enhances this method by integrating state-of-the-art deep learning techniques, particularly CodeBERT [16], and topic modeling, aiming for greater granularity in identifying software features and requirements.

Automatically identifying reusable features and their requirements can significantly facilitate a better understanding of software systems. This approach not only assists in comprehending the complexities of software requirements but also supports faster development, potentially minimizing risks compared to developing software entirely from scratch. However, it is essential to acknowledge that the effectiveness and applicability of reusing these features also depend on specific project requirements, system compatibility, and the quality of the existing codebase. Since the identified features have already been tested in a legacy application, they will likely be less error-prone than newly developed versions. Their streamlined representation further highlights the clarity of software requirements when mapped onto a feature model. The feature model, expressed through a feature diagram, clarifies features and their intricate inter-dependencies in a hierarchical tree structure [19], [20]. This model-centric approach to software development simplifies visualizing imperative features, encompassing aspects like testability, traceability, reusability, testing workflow, constraint verification, and validation. The feature model enables feature-to-artifact mappings to illustrate how system functions are implemented in terms of relationships, dependencies, behaviors, and constraints [21].

Next-generation embedded software: Next-generation embedded software represents the evolution of traditional embedded software, marking a significant transition in its capabilities, functionalities, and applications. It is characterized by its software-centric approach, incorporating advanced technologies such as artificial intelligence, connectivity, and advanced computing paradigms like fog and edge computing. These systems are distinct from traditional ones due to their ability to process complex tasks, adapt to rapidly changing technologies, and meet strict safety and real-time performance requirements. An example of next-generation embedded software is found in autonomous vehicles. These vehicles integrate complex AI algorithms for navigation and decision-making, process vast amounts of data in real time through edge computing, and require high levels of safety and reliability.

Additionally, with the increasing complexity of advanced features in next-generation embedded software, a feature-based modular embedded software design and development approach becomes vital to facilitate the integration of diverse applications seamlessly. This approach is particularly important in the creation of efficient computing architectures. For instance, autonomous vehicles incorporate diverse machine learning-based applications (e.g., video processing) that demand faster computation for making real-time decisions. As these machine learning models grow in complexity, there is a limitation in training them on resource-scarce embedded systems. Thus, the demand for a parallel and distributed framework to support these ML models arises, aiming to boost the training speed and process large chunks of data in real-time [22]. Furthermore, to address the concern for safe online training and inference, which is crucial in safety-critical systems, the Triple Modular Redundancy (TMR) technique is introduced. TMR ensures reliability and fault-tolerance in the system, as it takes three identical copies of a processor, and their outputs are then voted upon to produce a single output. This redundancy mechanism's primary benefit is detecting and correcting faults, adding an extra layer of reliability to the system.

Embedded systems, evident in technologies like autonomous driving and smart homes, are now integral to our lives. The call to merge these devices into a cohesive network has led to the rise of fog computing, which uses edge devices to process data closer to its source. This shift promises efficiency and real-time decision-making. Research highlights fog computing's aptness for supporting mobility, time-sensitive, and resource-heavy applications [23]. It offers better network reliability and reduced delays compared to traditional cloud models, marking a significant stride in the evolution of embedded systems. As fog computing becomes foundational for future software, this research emphasizes its crucial role for developers keen on staying updated with technological advancements [23].

The primary objective of this research is to address the identified gaps and establish a methodology that enables legacy embedded software code reuse, guarantees the integrity of configurations, and boosts application performance by channeling computations closer to data sources — the fog and edge networks. We are not just focusing on technical advancements but envisioning a paradigm shift. We propose a feature-driven approach to embedded software design to integrate state-of-the-art functionalities into the core software framework seamlessly.

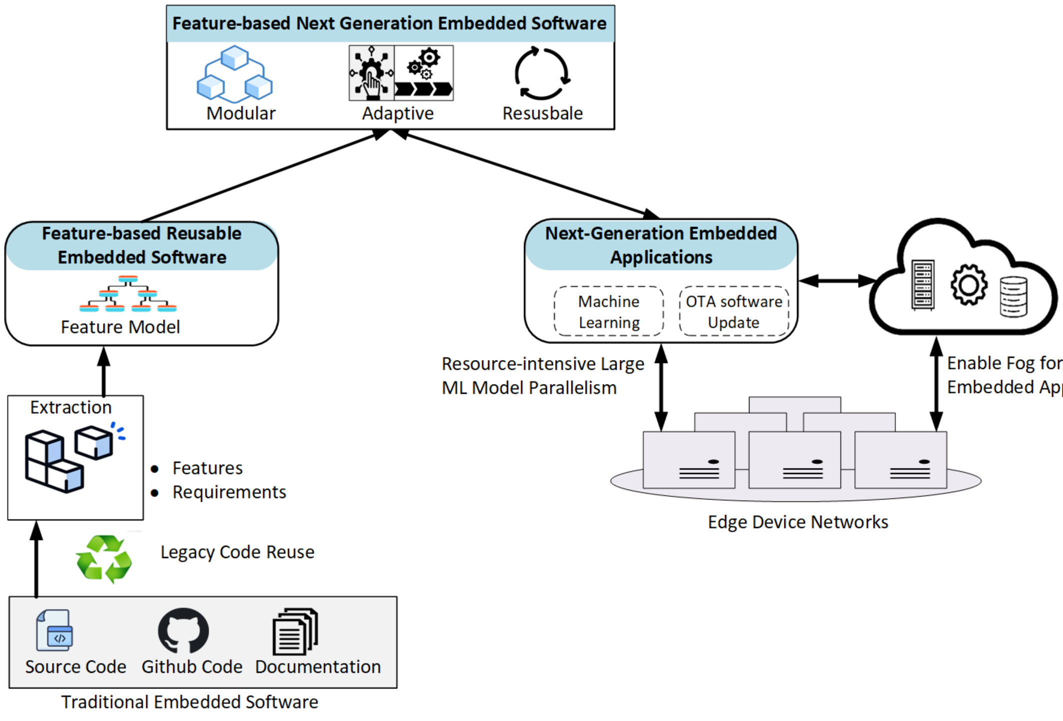

Feature-based Next Generation Embedded Software
Modular
Adaptive
Resusbale
Feature-based Reusable Embedded Software
Feature Model
Next-Generation Embedded Applications
Machine Learning
OTA software Update
Resource-intensive Large ML Model Parallelism
Enable Fog for Next-Gen Embedded Applications
Edge Device Networks
Extraction
Features
Requirements
Legacy Code Reuse
Source Code
Github Code
Documentation
Traditional Embedded Software

Figure 1.1: Workflow of our proposed feature-based embedded software design and development

To visualize our research's overall goals, we present an abstract representation of our workflow for feature-based embedded software design and development, showcased in Figure 1.1. This workflow outlines the systematic process our proposed system undergoes. Initially, relevant features and requirements are extracted from the existing codebase. Using this extracted information, we then formulate a comprehensive feature model, which is useful in analyzing and understanding the configurations of these features. In the subsequent phases, we design next-generation embedded applications like machine learning and OTA software updates. All of these advanced functionalities, treated as distinct features, come to life in a fog-enabled environment, ensuring fast and efficient management of advanced features.

1.1 Motivation

The complexity of next-generation embedded software continues to increase due to high variability, which is expected to grow further with the extensive integration of sensors, computing platforms, and constraints for ensuring safety, security, and quality. Consequently, the design and development cycles of such software take longer than usual to verify and validate its features. Embedded software manufacturing companies face the challenge of bringing products to market quickly to remain competitive.

This necessitates reducing design and development cycles by reusing legacy embedded software. However, developers face challenges in reusing code because embedded software possesses different characteristics than typical software products. As a result, developers often avoid reusing legacy embedded software, finding it hard to understand the required features and constraints without domain expertise. Numerous embedded software repositories are publicly available and hold potential for reuse in developing new applications.

One significant challenge is understanding the necessary features and their requirements due to a lack of domain knowledge and often insufficient documentation. Furthermore, designing next-generation machine learning applications brings its own set of challenges. These applications often come with unique requirements and are frequently structured in a non-modular manner. Such attributes compound the challenges of parallel and distributed computation, posing difficulties for seamless integration into existing systems. Transitioning to a feature-based embedded software design can be a solution, promoting a more structured and efficient development process.

Extracting information from existing or legacy embedded software will help developers visualize the software design and speed up the development process. This study aims to find a way to design and develop next-generation embedded software where developers can extract the required information for faster software development. Moreover, next-generation embedded applications demand comprehensive support to process large volumes of sensor data and machine learning model training to meet application constraints. This motivates the integration of a fog-enabled architecture to simplify the computing process and manage next-generation embedded applications more effectively.

In our ongoing endeavor to find a robust computing solution for next-generation embedded applications, we examine the prospects of fog and edge computing architectures. These architectures can benefit next-generation embedded software, enhancing mobility and response times by processing data closer to devices. Unlike cloud computing, fog, and edge computing reduce latency for real-time applications and supports dynamic environments where fast decision-making is critical. By decentralizing computation and networking, fog, and edge computing can meet the advanced demands of next-generation embedded systems, ensuring rapid, scalable, and efficient operation. A primary challenge in achieving this goal is seamlessly integrating ML-based embedded applications, like deep neural networks (DNNs), into embedded devices, which is made more difficult by the large size and complexity of DNN models. The conventional approach to adapting these large models in resource-constrained environments often needs to catch up, encouraging us towards innovative solutions like machine learning model splitting, a strategy that promises to accelerate performance substantially by distributing the computational load more evenly and efficiently across available resources.

The next-generation applications, characterized by their mobile and distributed computing nature, will be connected to multiple of computing devices, necessitating a seamless and efficient update mechanism. OTA updates are critical in this context as they enable remote software updating without the need for physical access to the devices. This method works by the application or device periodically checking for updates from a centralized server and, upon availability, downloading and installing these updates wirelessly.

During our investigation, we discover that FCA's automotive brands, such as Chrysler, Dodge, Jeep, and Ram, have received feedback from customers about serious glitches and long delays in the software update process [24]. As a result, we present a case study of OTA software update time estimation to help auto manufacturers and car owners decide on the best strategy. Although OTA software update frameworks are still in the early stages of development, automobile manufacturers need to be more proactive in finding an efficient OTA update framework to provide the best user experience for their customers.

1.2 Problem Statement

Developing next-generation embedded software from scratch is challenging due to the increased complexity and the need for a deep understanding of system input, output, architecture, and configurations of required features. To address this issue, our research focuses on the following key problems:

a) Reusable Features and Requirements Identification

- Identifying and extracting reusable features from legacy embedded software source code, which can be potentially repurposed for new applications, reducing development time and cost.
- Analyzing the relationships and requirements among reusable features to ensure that they can be effectively integrated and configured in a new system while maintaining the desired functionality.

b) Feature Model Creation

- Building a feature model with valid configurations to represent the extracted features and their relationships, aiding in the design and development process of new embedded system applications.

c) Designing a Fog and Edge-assisted Architecture for Satisfying Software Requirements

- Enabling a fog and edge computing architecture to meet the requirements of next-generation embedded applications, such as the need to execute resource-intensive machine learning algorithms and ensure faster OTA software update for real-time decision making.

d) Designing and Developing Feature-based Next-generation Embedded Software

- Transforming non-modular code to a feature-based modular structure to facilitate the seamless integration of advanced features in next-generation embedded systems, enhancing the efficiency and adaptability of the software.

Therefore, our research focuses on designing and developing feature-based next-generation embedded software to address the above problems, ensuring that the resulting systems are efficient, reliable, and capable of meeting the increasing demands of modern applications.

1.3 Research Questions

To enhance the reuse of code for designing and developing next-generation software, we need to minimize the uncertainty of our approach in satisfying application requirements (e.g., design, functional, and non-functional). Existing methods implemented by software engineering tools do not benefit developers in identifying requirements automatically from the embedded source code. Thus, our research will benefit from managing the complexity of different functionalities when developing embedded applications. To an extent, our study demonstrates the suitability of feature-based embedded software in fog computing architecture supporting advanced functionalities. To address the above objectives, we formulate the following research questions:

- *RQ1:* What are the essential criteria and information needed to identify reusable features from legacy embedded software?
- *RQ2:* What method can assist in reusing identified features, along with the creation of a feature model expedite the development process of next-generation embedded software?
- *RQ3:* What is the efficacy of BERT and topic modeling techniques in identifying and extracting features and requirements from embedded software's source code?
- *RQ4*: Is fog computing architecture effective in enhancing performance and satisfying the constraints of next-generation embedded applications?
- *RQ5:* What strategies facilitate the optimal training of large machine learning models for next-generation embedded applications, and what approaches ensure the safety of online model training during this process?

By addressing these research questions, we aim to contribute to the body of knowledge in the domain of next-generation embedded software and provide practical solutions for the design and development of feature-based next-generation embedded systems.

1.4 Contributions

The proposed framework for building feature-based next-generation software enables application developers to focus on reusing functionalities and addresses application requirements in the underlying computational fog infrastructure. To answer the research question *RQ1*, we extract information from function call graphs and source code using topic modeling and BERT to identify functional requirements. We also parse information from available documents such as developer comments, requirement specifications, and readme.md files to identify non-functional requirements. Addressing *RQ2*, we use the feature modeling process of the software product line method to reuse the identified features from legacy embedded software, which have already been tested or verified, thereby facilitating faster software development. Moreover, we compare our automatic feature identification with the manual and rule-based approaches to ensure its efficiency. For *RQ3*, we evaluate the efficacy of BERT and topic modeling techniques in extracting features and requirements from the source code of embedded software, comparing with manual and rule-based approaches.

To address *RQ4*, we implement an OTA software update case study utilizing fog architecture in estimating update time, showcasing the substantial reduction in update time under a fog computing architecture essential in supporting future real-time decision-making requirements. To tackle *RQ5*, we explore strategies like model parallelism and find optimal partitions for efficiently training large machine learning models utilizing available edge device resources. To ensure the safety of online model training, we use the TMR technique, thereby enhancing overall safe execution.

The main contributions of this research work are:

a) Feature Identification and Model Construction for Improving Embedded Software Reuse

- Employed Topic Modeling and BERT for identifying reusable features from legacy code and analyzed functional and non-functional requirements.
- Developed a systematic approach to build, validate, and visualize relationships between features in a comprehensive feature model. This model aids in visualizing and understanding the various software requirements.

b) Design and Development of Modern Performance Features for Embedded Applications

- Designed and developed a fog-enabled architecture to minimize latency in real-time applications and enhance overall performance,
- Implemented a real-world testbed architecture to integrate next-generation embedded application requirements, demonstrated through a case study on OTA software updates.
- Developed techniques for efficient large machine learning model training on embedded systems.

c) Feature-Based Approach for Next-Generation Embedded Software Development

- Enabled the transformation of advanced applications from a non-modular to a modular structure, enabling their reuse in feature-based software development.

1.5 Scope of the Research

In this research, we mainly concentrate on the publicly accessible legacy software that is written using C procedural and function-oriented language. The rationale behind this scoping is that our study primarily targets families of embedded systems that evolve from their ancestral legacy systems, which predominantly utilized the C language. The existing analyses show that most embedded legacy systems are written using C language because of the advantage of low-level implementation. Although we limit our approach to C-based software, this idea can be extended to other languages as we have used a function call graph for extracting reusable features. In addition, we make a necessary ground truth assumption for feature identification: the manual analysis of legacy code serves as the benchmark for determining the features in the embedded systems, forming the basis for evaluating our automated feature extraction methods.

To exhibit the versatility and applicability of the feature-based embedded software development approach in accommodating advanced features, we extend our considerations to non-structured Python programs. Here, we undertake the initiative to modularize these programs, promoting reusability and facilitating the creation of a feature model that effectively integrates advanced functionalities.

In this research, our primary focus is on embedded applications that necessitate rapid decision-making, especially in scenarios where latency plays a pivotal role. It is essential to acknowledge the inherent latency associated with cloud computing, which can introduce delays, potentially compromising the real-time response and efficiency of the system. Consequently, for the specific scope of our study, we have consciously chosen to prioritize fog and edge computing over cloud computing. By leveraging the capabilities of fog and edge computing, we aim to capitalize on their proximity to data sources, thus minimizing latency and ensuring faster data processing, which is paramount for the applications under consideration.

Furthermore, we envision fog computing as a promising infrastructure that addresses the prerequisites of advanced applications, including mobility support and ML-based computations. This foresight emphasizes the role of fog computing in enabling advancements in next-generation embedded software. An essential requirement, envisioned to become indispensable in forthcoming embedded software iterations, is the OTA update. Our exploration investigates the intricacies of OTA updates, aiming to seamlessly integrate them within the fog computing framework, emphasizing the importance of real-time updates and system efficiency enhancement.

1.6 Organization of the Thesis

The structure of this thesis is outlined into seven chapters, offering a comprehensive insight into the methodologies, applications, and evaluations undertaken. The evaluations of our proposed methodologies are presented separately within each chapter to enhance clarity and comprehension. This approach allows for targeted and contextual analysis of our proposed methodologies and simplifies understanding of complex concepts, making the thesis more accessible and impactful for readers.

- **Chapter 1: Introduction**

  This chapter introduces the foundational aspects of the research, highlighting the research questions and objectives. It also offers a brief overview of the subsequent chapters in the thesis.

- **Chapter 2: Background and Related Works**

  A detailed literature review is undertaken in this chapter, exploring the existing methodologies, challenges, and advancements relevant to embedded software, software reuse, and fog computing. This chapter sets the theoretical backdrop for the ensuing research chapters.

- **Chapter 3: Feature-based Reusable Embedded Software Design**

  This chapter focuses on our proposed methodologies for extracting reusable features and requirements from legacy embedded software. It delves into the use of advanced NLP techniques like BERT and topic modeling, essential for modern software reuse practices. It is divided into two sections:

  - *Feature and Requirements Extraction:* Techniques utilizing BERT and topic modeling to extract and analyze features from legacy code are elaborated upon, emphasizing the role of these insights in software reuse.
  - *Feature Model Construction and Validation:* The section focuses on the construction of a feature model tailored for embedded systems and its subsequent validation.

- **Chapter 4: Next-Generation Embedded Applications Design**

  This chapter, segmented into three primary sections, dives deep into the integration of advanced technological facets into embedded software:

  – *Enable Fog Computing Architecture for Next-Gen Embedded Software:* Focuses on deploying the fog-enabled software and its performance evaluation, with special emphasis on OTA software updates.

  – *ML-based Fog/Edge Assisted Software Development:* Discusses the integration of machine learning application and conceptualization of a parallel model geared towards enhanced training and inference.

  – *Safe Online Model Training:* Introduces the TMR technique, underscoring its significance in ensuring online model training's safety.

- **Chapter 5: Feature-based Next-Generation Embedded Software Development**

  This chapter explores how the advanced applications (e.g., machine learning)) code can be reused in the form of feature tree structures. The emphasis is on the structured development and integration of these features in a feature model, facilitating enhanced system performance.

- **Chapter 6: Results Analysis and Discussion**

  It presents the additional results from experiments conducted following our proposed methodologies and clarifies our research questions for this study.

- **Chapter 7: Conclusion**

  Provides a summative discussion on the research findings, concluding remarks, and potential avenues for future research.

Mapping Chapters to Associated Publications: The list of associated publications of the following chapters, demonstrating the research journey and its dissemination through academic channels:

- **Chapter 3:**

  - WiP ICCPS 2023 [25]: Highlighted automated feature identification for Cyber-Physical Systems using topic modeling.

  - IEEE Access 2022 [3]: Discussed how to improve the reuse of functions in embedded software.

- **Chapter 4:**
  - IEEE ICMLA 2023 [26]: Discussed safe online ML model training and inference on edge networks.
  - Canadian AI 2023 [27]: Focused on optimizing DNN model partitioning for enhanced performance.
  - Book Chapter Springer 2022 [28]: Covered cognitive mobile computing and communication technologies for Cyber-Physical Systems.
  - IEEE TSC 2021 [29]: Demonstrated faster fog computing based OTA vehicular updates.
  - IEEE ISNCC 2020 [30]: Addressed resource-efficient allocation of fog nodes.

Chapter 2

Background and Related Work

2.1 Background

Next-generation embedded systems are increasingly software-intensive to meet advanced customer requirements. They demand high processing and storage capabilities to support low latency, real-time embedded applications. Furthermore, embedded system applications are becoming more distributed, heterogeneous, and complex. This necessitates designing decentralized variability management solutions using Cloud-Fog-Edge computing architecture. To assist in designing and developing next-generation embedded software, our research focuses on two important issues that need to be addressed with high priority.

The research identifies gaps in existing software design and development mechanisms:

- Identifying the complex requirements of various needed functionalities for accelerated application development.
- Addressing the high-level dependability requirements of any computing architecture to meet application requirements (e.g., real-time processing).

In software engineering, the systematic steps required for software development follow as i) requirements analysis, ii) design, iii) construction, iv) testing, and v) maintenance. Therefore, software development becomes time-consuming when essential information is unclear or absent for software designers and developers. In particular, embedded software development is more challenging than traditional development due to the lack of domain knowledge and additional constraints. Thus, we analyze existing studies to reuse legacy software for faster development with the necessary functionality. We find software product line and reverse engineering approaches as potential solutions for quicker development and management of embedded software. Along with this, we need to identify an efficient architecture for managing the computational requirements of embedded applications. We investigate cloud and fog computing architectures to better fit the requirements of next-generation embedded software. In this section, we discuss the basic terminologies and related works to understand our investigations in developing next-generation embedded systems.

2.1.1 Reverse Engineering

The traditional development process follows forward engineering, a method for constructing an application based on specified requirements. This approach can be costly due to the high proficiency skill requirements and time-consuming nature of testing new applications.

In contrast, reverse engineering [31] is the opposite process of application development, where an existing application (e.g., source code) is analyzed to extract knowledge or understand its architecture. This method takes less time to develop an application and minimizes potential risks associated with writing software from scratch. Since the extracted features have already been tested for a given application, they are likely to be less error-prone than newly developed versions. As a result, the development of embedded system applications can be accelerated by employing reverse engineering techniques. Figure 2.1 shows the difference between forward and reverse engineering process.

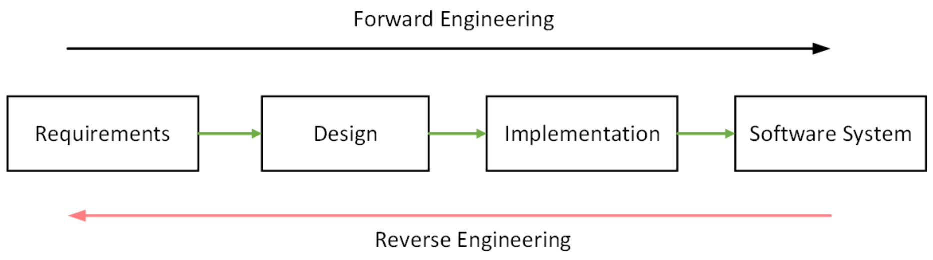


Figure 2.1: Forward vs reverse engineering

2.1.2 Static Code Analysis

Static code analysis refers to the technique of evaluating a program's source code without executing it. This method is primarily used to uncover potential vulnerabilities, bugs, or violations of programming conventions in the code. In the context of embedded software, through static code analysis, we extract program information to understand the system design constraints and real-time requirements accurately.

The *cflow* Utility:

The *cflow* utility is a classic tool used in static analysis to visualize a program's control flow. By generating a call graph that outlines function calls and their hierarchy, cflow provides a snapshot of how different embedded software modules interact.

Topic Modeling

Topic Modeling is a type of statistical model used to uncover abstract topics within a collection of documents. In the context of static code analysis, these "documents" can be perceived as different source files, modules, or code artifacts of the software. Topic modeling shows immense promise in the semantic analysis of code. By identifying patterns and recurring topics within the code, it can highlight areas of the software that handle specific functionalities. This becomes invaluable when trying to understand or modify features in a large or unfamiliar codebase.

TF-IDF: Term Frequency-Inverse Document Frequency (TF-IDF) is a numerical statistic that reflects how important a term is to a document within a collection. It considers both the frequency of the term in a particular document and its frequency across all documents.

LDA: Latent Dirichlet Allocation (LDA) is a popular method for topic modeling. It assumes each document is a mix of topics, and a topic is a mix of words. By analyzing the distribution of words across documents, LDA can deduce the topics present in a corpus. By employing tools like TF-IDF and LDA, one can understand the primary functionalities of different software parts, which becomes especially useful when dealing with expansive codebases or when trying to decode legacy code.

BERT Model

Bidirectional Encoder Representations from Transformers (BERT) [32] is a deep learning model designed to understand the context of words in a sentence. It is pre-trained on a large corpus of text and can generate embeddings for given input text. Unlike traditional models that analyze words in a sequence either from left to right or right to left, BERT considers both directions, capturing each word's context more holistically.

Embeddings: In natural language processing (NLP), embeddings are dense vector representations of

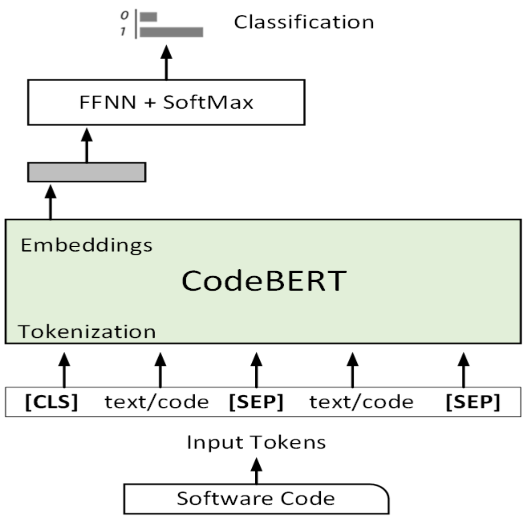


Figure 2.2: BERT model architecture [33]

words, sentences, or even paragraphs. They capture the semantic information, meaning that similar words or phrases will have embeddings that are close to each other in the vector space.

For static code analysis, CodeBERT or CodeBERTScore (extension of BERT) shines in its ability to understand the semantic nuances of the code. When analyzing function calls or variable assignments, BERT's contextual grasp can be used to understand constraints or intricate functionalities. Its strength lies in its bidirectional nature, which allows for a deep understanding of the relationships and dependencies within the code.

- **Impact of CodeBERT on embedded software source code analysis:** CodeBERT's contextual capabilities can lead to nuanced insights, enabling developers to identify hidden dependencies, predict potential issues, and understand intricate code functionalities.

- **Impact of Topic Modeling on embedded software source code analysis:** Topic Modeling can illuminate areas of the code that are dedicated to specific functionalities, helping in modular understanding and efficient feature addition or modification.

2.1.3 Software Product Line

A software product line (SPL) [34] refers to a systematic approach for designing and developing new software systems by reusing legacy software assets. It aids in increasing productivity and software quality, as it reuses already implemented legacy software features. Consequently, the time for feature verification and validation is reduced. The product line approach presents a feature model representing feature commonalities and variabilities. However, due to the rapid growth of advanced features and their evolution, the feature model frequently changes. Therefore, a reverse engineering process is needed to maintain consistency between available feature configurations and generated feature models. Software product line development is divided into two phases [35]:

- *Domain engineering:* This phase specifies software requirements such as the required feature list, feature relationships, feature constraints, and so on. The outcome of this phase includes extracted information for developing software from reusable assets, such as required features, software architecture, feature model, and software components.
- *Application Engineering:* In this approach, engineers select the required features to build a new application with different requirements.

Feature Model for Requirements Analysis

The Feature Model (FM) is a fundamental artifact of the Software Product Line (SPL) [36]. It organizes features in a tree where features are represented in parent-child relations to define the concept of an application family. The parent features are linked to child features using edges, which represent containment relationships [35].

- In a feature model, features can be categorized into two types:
  - *Mandatory:* A mandatory feature must be present in all applications of the product line.
  - *OR:* The OR feature group selects at least one feature from its child feature list. The selection cardinality is (1 to k), where k is the group size.
- When features are represented in a group, the feature model defines two feature groups [37]:
  - *Alternative (XOR):* In XOR, features are selected from the set of child features following the cardinality constraint. For example, one parent feature selects any one child feature (1 to 1).
  - *OR:* The OR feature group selects at least one feature from its child feature list. The selection cardinality is (1 to k), where k is the group size.

- A feature model also illustrates two types of constraints in feature selections for valid configurations.
  - *Requires:* The *requires* constraint means that if a feature f_1 is selected, another feature f_2 must be selected.
  - *Excludes:* Conversely, the *excludes* constraint states that if the feature f_1 is selected, then the feature f_2 cannot be selected.

2.1.4 Decentralized Computing for Embedded Systems

Next-generation embedded systems rely on decentralized computing infrastructure, as they require collecting and processing massive amounts of sensor data. Decentralized computing implies that computing resources are located between the data source and the cloud/fog/edge or other data centers. Modern embedded system applications can leverage cloud-based solutions to support resource management [38], data storage, security, and access to various software applications. However, cloud-based approaches are not always feasible for applications with constraints such as response time and cost. Moreover, cloud service is not always the preferred solution considering downtime, communication delay, privacy, limited control, and flexibility [39].

Therefore, fog computing may be advantageous for running time-sensitive applications on the nearest computing platforms (e.g., vehicular fog computing, roadside units (RSUs)), providing better performance. Furthermore, several issues need to be investigated for these fog computing platforms, such as hardware architectures, run time, communication mechanisms, wireless networking, and application development on constrained devices. These investigations will help applications run seamlessly across diverse platforms [40].

Wireless Communication Technologies: For next-generation embedded software, wireless network communication technologies are becoming popular due to their improved usability in managing large networks. To visualize the differences among common wireless technologies, a comparison table can be presented (Table 2.1). This table compares various metrics such as standard, data rate, frequency, operating range, latency, and application area.

2.1.5 Parallel Computing for Machine Learning in Embedded Systems

The current software design for embedded systems requires the adoption of parallel computing to benefit from the presence of multiple processors. Traditionally, machine learning-based embedded applications were executed on a single-core processor with sequential execution of tasks [42]. However, this approach is no

Table 2.1: Comparison of wireless technologies in fog computing [41]

Technology	**Standard**	**Max Operating Range**	**Data Rate**	**Latency**	**Frequency**	**Potential Use Cases**
Zigbee	IEEE 802.15	100m	250Kbps	<140ms	2.4GHz	Wireless Sensors (Monitoring)
Bluetooth	IEEE 802.15	200m	1 to 3Mbps	100ms	2.4-2.5GHz	Wireless Sensors (Monitoring, Control)
WiFi	IEEE 802.11	70-250m	Upto 600Mbps	<1ms	2.4-5GHz	Mobile Internet
4G	IEEE 802.16e	10 miles	Upto 1Gbps	10ms	<6GHz	Streaming Video
5G	3GPP	1500 feet	Upto 20Gbps	<1ms	30-300GHz	IoT, VR
DSRC	IEEE 802.11p	<100m	27Mbps	150ms	<6GHz	Vehicular networks
Real-time WiFi	IEEE 802.11 b/g/n	70-250m	150Mbps	<4.2ms	2.4GHz	Wireless control systems
C-V2X	C-V2X Rel. 16	107m	4 to 50Mpps	<50ms	<6GHz	Vehicular networks

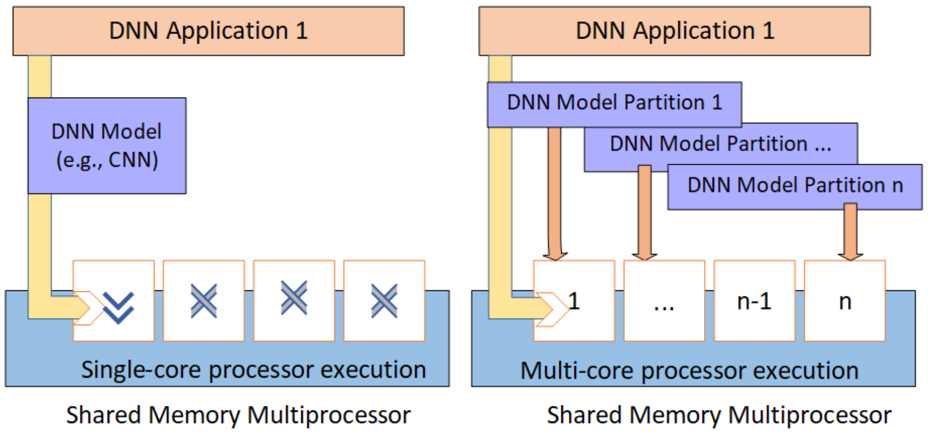


Figure 2.3: Architecture of multiprocessor systems for DNN applications

longer entertained due to the increasing demand for faster computations in various modern applications. To take advantage of the multiprocessor devices or general-purpose computing systems (CPUs) [43] in these systems, the applications must possess adequate parallelism characteristics that enable their computation across different processors or devices. In the industry, the next generation of embedded systems must improve their performance by parallelizing the ML model to achieve faster training and response times. Therefore, we see an opportunity to partition the ML model, such as a DNN, across multiple processors for faster computations.

DNN Application on Multiple Cores:

In conventional embedded software design, the execution of ML application tasks typically runs sequentially unless specifically designed for parallel processing. As a result, available computing processors or cores remain underutilized, while modern ML applications could benefit from parallel execution on multi-core processors. Figure 2.3 illustrates the difference between single-core and multi-core processor execution for deep neural network (DNN) applications. With regards to embedded systems design, Electric Control Units (ECUs) are increasingly adopting many-core processors to accommodate a growing number of functionalities. However, partitioning the application's tasks, such as the ML model or workload, can significantly decrease execution time and enhance overall resource utilization. Model parallelism has a substantial impact on the acceleration of parallel execution.

AlexNet: AlexNet is one of the famous CNN architectures in DNN. Alex Krizhevsky *et al.* [44] designed this network, which dramatically impacts parallel computation in the field of deep learning applications. The proposed network includes five convolutional layers following max-pooling operations and three fully-connected layers. The architecture adds a dropout layer after each dense or fully connected layer. In addition, the use of rectified linear unit (ReLu) activation function after each convolution and dense layer controls the exponential increase of computation. However, the output layer runs the softmax activation to predict the output classes' probability distribution. The CNN model input image size is $227 \times 227 \times 3$. The AlexNet architecture processes around 1.1 billion computation units in a forward pass for 62.3 million parameters [45]. Although the convolution layers hold only 6% of all parameters, they consume 95% of the total computation.

Data Parallelism: Data parallelism is one way of performing parallel execution of different tasks across multiple processors. In this approach, a machine-learning model will be replicated on different computing nodes. Then, the input data is split into different batch sizes for concurrent execution. Each worker node must synchronize all the model parameters or gradients to ensure consistent model training. This approach is efficient when the computation of each weight becomes high and the training time increases for processing the larger dataset [46].

Model Parallelism: Model parallelism occurs across the model dimensions where different workers compute distinct parts of the model concurrently. To implement model parallelism, the partitioned layers must perform independent tasks, and the workers need to synchronize their executions. Traditionally, model parallelism is suggested when a single processor can not run a large ML model due to memory constraints. This approach is beneficial when the computation per neuron becomes high [46]. As a result, the model gets split to fit the model's parameters in memory. However, this approach has a complex implementation, as there need to be optimal splits with correct resource assignment.

Pipeline Model Parallelism: Although model parallelism is used to partition the larger model into different computing processors, it might suffer from resource underutilization when processors depend on each other for their inputs. As a result, processors remain idle. In such a situation, pipeline model parallelism splits the input data into multiple batches and makes a synchronous pipeline execution so that all the processes can continue training with the subsequent available input batches. A Gpipe [47] pipeline concept is introduced by training large-scale neural network layers on separate accelerators.

Current Deep Learning Frameworks: To support the DNN model training, we find different deep learning frameworks that use data and model parallelism techniques for faster computation. For example, TensorFlow, PyTorch, MxNet, and Caffe are the most popular distributed frameworks in recent time [48]. Although Tensorflow supports both data and model parallelism, it lacks in running the different numbers of workers on particular devices for implementing communication scheduling. PyTorch is a successor of the Caffe framework and provides high-speed computation by using a dynamic computation graph. The recent DNN applications can be benefitted by using its model parallelism feature. However, the framework is still a work in progress [49].

The landscape of software engineering has witnessed considerable advancements in areas like feature identification, traceability, deep learning, and topic modeling. Each of these domains, while diverse, offers unique perspectives and methodologies. In this section, we investigate the complexities of these areas, paying particular attention to the evolving connection between fog computing architecture and machine learning applications. Additionally, we examine the implications of software modularity, adaptive feature modeling, feature extraction, and the pivotal role of security in the realm of embedded software development. This section provides a comparative overview of related work in these domains, emphasizing methodologies and the challenges they address.

2.2 Related Work

Feature identification, traceability, deep learning, and topic modeling have been extensively studied in software engineering, often with different goals and approaches in mind. This section provides a comparative overview of related works in these domains, emphasizing methodologies and the challenges they address.

2.2.1 Traditional Approaches for Features Identification from Source Code

Legacy Techniques and Their Limitations: Many software engineering techniques have been extensively used for identifying reusable units and requirements for developing an application from legacy code. However, these techniques often require requirements to be provided upfront or manually specified by users [5], [50].

Clone and Own Approach: Stefan Fischer *et al.* [51] present an Eclipse-based framework that supports the "clone and own" practice by automatically extracting reusable features from Java applications. The clone and own process consists of three steps: extraction, composition, and compilation. The authors propose automated extraction and composition to assist developers in identifying potential artifacts from previously developed products, mapping relationships among those artifacts, and storing this information in a database for composing a new product.

Reverse Engineering with Algorithmic Identification: In [12], a three-step process is introduced to identify features from the source code of product variants using reverse engineering. First, the product model is reverse-engineered from the source code to reduce noise caused by different implementations of the same feature. Then, each product model is divided into a set of smaller pieces. In the second step, an algorithm is implemented to identify identical features. In the final step, non-relevant candidates are manually eliminated, and missing features are added if necessary.

Concern-Driven Software Development: In another work, Omar Alam *et al.* propose [52] a concern-driven software development approach in which they create a feature model considering the variability of the interfaces of the concerns (e.g., units of modularization) instead of focusing on the functional definition of any component.

Model-based Adaptive Development: The authors of paper [53] present a context feature model-based adaptation logic architecture, simplifying self-adapting system development. The method integrates knowledge of system state and operating context, allowing more accessible analysis and planning of reconfigurations. The work is validated via application to the Tasklet system for mobile code offloading. In another work, S. Umair *et al.* [54] propose a novel Model Driven Reverse Engineering (MDRE) framework called Src2MoF to generate Unified Modeling Language (UML) structural and behavioral diagrams from Java source code. Moreover, In a paper [55], Eunsuk *et al.* introduce a model-based framework for analyzing security configurations in systems with complex structures, recognizing misconfiguration as a leading cause of security failures. This framework is founded on a three-fold approach, involving the collection and representation of domain knowledge by an expert, security property specification by the end user, and an analysis by a test engineer.

Extending Feature Models: Jessie *et al.* [56] highlight the limitations of basic feature models, also called boolean feature models, which only represent boolean features. This means they only identify if a characteristic is present or not in a software system. The authors propose an extension of the existing boolean variability model that considers multivalued attributes or UML-like cardinalities to support variability modeling in complex product lines.

Table 2.2: Comparison of different reverse engineering approaches for identifying features

Paper	**Idea**	**Techniques**	**Inputs**	**Output**	**Language**	**Tools Used**	**Use Code Repository?**	**Case Study**
[12]	Semi-automated feature identification from source code	Product Line Engineering	Source code	Feature model	Object Oriented (e.g., Java)	ExtractorPL	No	Banking Software Systems
[57]	Extracting features from models for SPL	Model-driven (MoVa2PL)	Set of model variants	Feature, Feature model	Object Oriented (e.g., Java)	ArgoUML	No	Entertainment system
[9]	Domain features identification	Natural Language Processing (TDM)	Source code, query input (e.g., Term)	Features	C	Fex, grep	Yes	Tools written in C (e.g., inotify-tools)
[54]	Generating high level UML models	Model Driven (MDRE), Text-to-Model (T2M)	Source code	UML model	Object Oriented (e.g., Java)	UML generator	No	ATM, Amadeus Hospitality
[58]	Feature Mining from legacy software	Conditional Probability	Feature model, Source code	AST	Object Oriented (e.g., Java)	Loong	Yes	Prevayler2, MobileMedia
Our Approach	Extracts highly reusable features	Function Call graph, Static Analysis	Source code	Reusable function, Feature Model	C	cflow, pycparser	Yes	Software controller for vehicles, Elevator system, Health monitoring

Feature Extraction from Source Code: To compare our proposed approach with existing related works, we present a comparison table (Table 2.2) specifying the research idea, input, output, techniques, language dependency, and application domain. We find that most related works focus on object-oriented software (e.g., Java), but P. Muller *et al.* [9] demonstrate how to extract features from C source code against a user-given query input (e.g., related term) employing natural language processing.

2.2.2 Advanced Methodologies in Feature Identification and Traceability

Topic Modeling in Software Traceability: A standout work by Asuncion *et al.* [59] highlighted the application of topic modeling in enhancing software traceability. Their approach, which relied on integrating topic modeling with both retrospective and prospective traceability, has proven to be effective, especially in the ArchStudio 4 software project.

Semi-Automated Feature Traceability: Another significant contribution in this domain is by Abukwaik *et al.* [61], which introduced a semi-automated feature traceability approach. By ingeniously combining

Table 2.3: Comparison of advanced methodologies in feature identification

Paper	**Problem Addressed**	**Limitation Addressed**	**Proposed Method**	**Technique/Algorithm Used**	**Domain/Area**	**Evaluation Method**
[12]	Feature identification in product variants	Tedious manual identification	Three-step approach: reverse engineering, algorithmic identification, manual pruning	Algorithm based on identifying identical components	Software Product Lines	Three experiments: Toy Banking, GPL and ArgoUML
[5]	Locating specific features in source code	Poor documentation and understanding of feature implementation	Semi-automatic technique combining dynamic and static analyses	Concept Analysis	Software Maintenance and Program Comprehension	Two case studies: web browser and commercial system
[59]	Enhancing software traceability	Difficulty in tracing links across heterogeneous artifacts	Integration of topic modeling with retrospective and prospective traceability	Latent Dirichlet Allocation, ACTS (Automated Component Trace capture System)	Software Engineering, Traceability	Case study on ArchStudio 4, feature comparison, timing and accuracy results for LDA
[60]	Locating features in software models	Difficulty in software maintenance and evolution in MDE	Integration of topic modeling into a genetic algorithm	Latent Dirichlet Allocation (LDA)	Model-Driven Engineering (MDE)	Two industrial case studies in railway and video game domains, compared with LSI baseline and Random Search sanity check
[61]	Tracing features in source code through embedded annotations	Manual tracing of feature locations in codebase	Semi-automated system using machine learning for feature traceability with embedded annotations	k-Nearest Neighbor (kNN), Support Vector Machine (SVM), Decision Tree (DT)	Software Engineering	Experiment with Clafer tools up to release 0.3.5, measuring performance in terms of precision, recall, and F-measure
[62]	Automated feature extraction from SRS documents for SPL	Time-consuming and error-prone manual feature extraction	Text preprocessing followed by NLP techniques to extract requirement sentences	POS tagging patterns and word dependency parsing rules using SpaCy	Software Product Lines (SPL)	Precision and recall metrics based on TP, FP, and FN counts
[63]	Timing analysis of embedded software updates	Lack of timing analysis	Update mechanisms analysis	Timing analysis techniques	Embedded Systems	Accuracy, performance
[64]	Onboarding challenges in SPLs	Lack of tools and documentation for newcomers	Interactive Concept Maps	Visual representation with interactive navigation	Software Product Lines	Online survey with Likert scale rated questions and the Technology Acceptance Model
[65]	Enhancing creativity in Requirements Engineering	Limited creativity approaches focused on exploring known possibilities	Automated framework for generating innovative requirements through combinational creativity	Social network analysis, Latent Dirichlet Allocation (LDA), part-of-speech tagging	Software Systems (Requirements Engineering)	Application on two open-source software systems (Firefox and Mylyn), Human subject evaluation
[66]	Manual feature location	Time-consuming, error-prone	Survey	N/A	Software Variability	N/A
[67]	Creation of high-quality, large-scale source code dataset	Privacy concerns due to the presence of PII in code	development of 'StarEncoder' to detect and redact PII in code	Bi-directionally self-attentive Transformers	Code Analysis	Evaluation of recall and precision for PII detection, comparison against regex baseline
Our Approach	Identification of CPS software features and requirements	Lack of complete documentation and domain knowledge in legacy software	Integration of BERT with topic modeling	BERT (specifically, CodeBERT), TF-IDF, Latent Dirichlet Allocation (LDA)	Cyber-Physical Systems (CPS)	Precision, recall, F1-score, execution time, sensitivity analysis

annotations in source code with machine learning techniques such as k-Nearest Neighbor (kNN), Support Vector Machine (SVM), and Decision Tree (DT), they managed to demonstrate impressive precision, recall, and F-measure results.

Morever, software modularity and feature extraction are foundational aspects of our work. One particularly relevant research is by Nadi *et al.* [68], where they mined configuration constraints for variability in

Linux build systems. While their work substantially contributes to understanding the variability in a large-scale system, it does not cater to automatic feature extraction from existing codebases, a gap addressed by our proposed framework. In another work, Paulius *et al.* [69] introduce a novel approach for automatically extracting features and generating feature models directly from Java programs. By using static code analysis and machine learning techniques, this method effectively identifies and models software features, thereby facilitating a more accurate understanding of the software and its variability. However, it does not account for security considerations, which could be a potential area for future research. The paper's contribution lies in the automation of a significant part of the feature modeling process, potentially boosting the productivity of software engineers and the overall efficiency of the software development process.

NLP for Requirements Extraction: Furthermore, Haris *et al.* [62] tackled the challenge of extracting features directly from Software Requirements Specification (SRS) documents, emphasizing that SRSs are more reliable for feature extraction. Their methodology, which heavily relied on Natural Language Processing techniques, showed significant success in extracting relevant features from SRS documents, especially given their high precision and recall values.

Topic Modeling and Feature Location: Perez *et al.* [60] delved into the domain of model-driven engineering (MDE), investigating the application of topic modeling for locating features in software models. Their method's robustness was demonstrated through the integration of Latent Dirichlet Allocation (LDA) with a genetic algorithm and evaluation using two industrial case studies.

Modern Tools and Datasets for Code Analysis:

Creation of the BigCode Dataset: In the context of creating datasets, the work titled "StarCoder: May The Source Be With You!" by Li *et al.* [67] takes center stage. The authors constructed the 'BigCode' dataset by harnessing source code from various public repositories and meticulously ensuring its quality.

Timing Aspects in Embedded Software: In another paper [63], the authors analyze the timing aspects of updating embedded software. It is particularly relevant to CPS as many CPS involve embedded software. While not directly linked to feature identification, understanding timing aspects is important for comprehending the non-functional requirements and constraints in CPS software.

Comparative Analysis:

When contrasting these methodologies, there's a clear presentation between works that focus on feature extraction from source code, such as Ziadi *et al.*[12] and Eisenbarth *et al.*[5], and those that emphasize traceability, like Asuncion *et al.*[59] and Abukwaik *et al.*[61]. Furthermore, while some research leverages topic modeling as a core methodology, such as Perez *et al.*[60] and Bhowmik *et al.*[65], others prioritize Natural Language Processing techniques, as exemplified by Haris *et al.*[62].

In summary, while there are a lot of methodologies and tools available in the literature to address feature identification and software traceability, our approach is unique in its emphasis on CPS software reuse. By seamlessly integrating static code analysis with topic modeling, we aim to comprehensively address the challenges inherent in this domain. Table 2.3 provides a comparative analysis of the discussed methodologies based on their core techniques, evaluation metrics, and main contributions.

2.2.3 Fog Enabled Distributed Computing for Embedded Systems

The increasing adoption of distributed processing devices is driving the growing demand for next-generation embedded software solutions. Many researchers and automotive industries are exploring various architectures to find solutions for next-generation embedded software, such as OTA updates. For example, Steger *et al.* [70] propose a framework for efficient and secure automotive wireless software updates in electronic control units. The framework employs IEEE 802.11s for vehicle-to-vehicle wireless communication and considers the entire life span of a vehicle, including development, assembly, maintenance, and service centers. To accelerate the software update process, the framework introduces parallel processing, allowing vehicles to update simultaneously, and partial software updates, where only the new part of the software is installed. Authentication and encryption are used for secure wireless software updates and data transfer. This framework primarily deals with software updates performed within a local network. However, autonomous/smart vehicles may also require substantial updates from remote data centers. The performance of this framework is assessed through real-world experiments using the Volvo ECU. Experimental results show that the required update times vary for different software sizes, ranging from 6.77s to 33.19s for software sizes between 67KB and 375KB.

Mirfakhraie *et al.* [71] propose a firmware update process for electronic control units based on firmware-over-the-air (FOTA) technology for commercial vehicles. FOTA is an innovative mobile software management technology developed by the telecommunications industry for remotely updating systems. Cellular devices use FOTA to update their products and fix bugs. Using cellular wireless technology, firmware updates are downloaded from remote data centers to the vehicle connectivity module. This approach helps avoid annual vehicle recalls, as updates are done periodically, reducing costs for customers, manufacturers, and dealers. Before provisioning a remote software update, this framework utilizes the MCC fleet tracker for information such as vehicle location, speed, and signal strength.

The OTA framework enables remote software updates but also exposes vehicle systems' vulnerabilities to the world. Tesla's Autopilot [72] assists in navigation and transportation by combining artificial intelligence and hardware technology. Tesla Autopilot aims to reduce driver fatigue and potential accidents on the road, requiring continuous remote updates for various software. Shantanu Ingle *et al.* [73] discuss different pitfalls of OTA updates and propose ways to make them safer.

Jan Bauwens *et al.* evaluate the overall feasibility of OTA updates, focusing on security, bug fixes, and software extensions [74]. A step-by-step approach is proposed to integrate software updates and assess the energy cost of each step. We also observe the use of Deep Neural networks in fog resource management [75], communication delay prediction, and other aspects of vehicular fog computing. For example, Salman Memon [76] *et al.* propose a machine learning algorithm to minimize the number of handovers in fog computing. The algorithm combines long short-term memory (LSTM) cells capable of learning the latency/cost associated with fog service requests. A machine learning-based cost predictor is employed to select the best-serving fog node in terms of cost and location. Additionally, Shen Wang *et al.* [77] present a transfer learning-based smart software-defined security (SSDS) mechanism, addressing the complex and dynamic security update problems in vehicle-to-grid (V2G) computing. Software-defined network (SDN) technology is adopted to implement the dynamic architecture for V2Gs.

While the majority of existing works primarily focus on maximizing network energy efficiency, our research investigates optimal resource allocation for fog nodes with the aim of minimizing software update time. Additionally, we employ transfer learning to predict network communication delay and demonstrate the impact of propagation and handover delay on OTA update time. Our approach utilizes machine learning to optimally distribute fog nodes within a cluster and leverages transfer learning to predict the communication delay between vehicles and fog nodes. We compare the predicted delays across two datasets: WiFi hotspot and 5G network datasets. A handover optimization algorithm is also proposed.

In compare to others, Our overall approach is implemented through a case study simulation in Mininet-WiFi [88]. To gauge the real-world implications of our proposed method, we design and implement a testbed platform that models a realistic environment. Our prototype employs QEMU and Uptane frameworks [89] to assess the feasibility of our approach in an actual setting. Furthermore, we compare recent works in Table 2.4 to better understand the current frameworks supporting next-generation applications.

Table 2.4: Comparison of different evaluation factors in the fog computing architecture

Work	Description	Communication Delay	Resource Provisioning	Mobility	Scalability	Technologies
Deze Zeng *et al.* [78]	Design an efficient task scheduling and resource management strategy with minimized task completion time	✓	✓	✗	✗	SDN
Tuli *et al.* [79]	It facilitates to IoT-Fog-Edge-Cloud integration offering execution of embedded applications at a time.	✓	✗	✗	✓	Raspberry Pi, Blockchain, Wi-Fi
Maheswaran *et al.* [80]	Proposed a framework based on fog computing to implement autonomous driving.	✗	✗	✓	✓	JAMScript, 5G Network
A A Khan *et al.* [81]	Present next-generation VANETs architecture analyzing delay, throughput, overhead.	✓	✗	✓	✗	SDN, C-RAN, 5G
R Mahmud *et al.* [82]	A simulator to develop models for service migration, distributor cluster, microservice orchestration for Edge/Fog computing.	✓	✓	✓	✓	iFogSim2
Andrei *et al.* [83]	A VANET system based on multilevel mobile edge computing that proposes latency-aware data offloading.	✓	✗	✓	✗	SDN, 5G
J Zhu *et al.* [84]	A fog computing framework named PEN (Phone+Embedded board+Neural compute stick) for the guide dog robot system.	✓	✗	✓	✗	Raspberry Pi, Image Processing, Neural Network
Suryadevara *et al.* [85]	A fog-based ubiquitous recognition model to assist adults suffering from dementia.	✓	✗	✗	✗	OpenHAB, Raspberry Pi, WSN
Jha *et al.* [86]	Present a simulator to model the behaviour of heterogeneous IoT and edge computing architecture	✓	✓	✓	✓	IoTSim-Edge/ CloudSim simulator, 5G
G. Kim al. [87]	Design architecture on data and service integrity for over-the-air updates	✗	✗	✗	✓	Blockchain, LTE, 4G, Wi-Fi
Mirfakhraie *et al.* [71]	Propose a firmware update process for an electronic control unit	✓	✗	✗	✗	FOTA, cellular

2.2.4 Distributed Machine Learning on Fog/Edge-Enhanced Embedded Systems

There is an increasing demand to integrate capabilities like machine learning to unleash the full potential of advanced features in embedded systems. However, such sophisticated features often demand parallel or distributed computing architectures to achieve optimal performance. The Fog/Edge computing paradigm has emerged as a promising solution to serve these needs, providing a distributed environment closer to data sources. Within this context, several strategies have been explored to accelerate the training of deep neural network models in distributed settings.

In recent years, various pipeline-based model parallelism techniques have been proposed to accelerate the training of Deep Neural Network (DNN) models. Examples of such techniques are PipeDream-2BW [90], PipeDream [91], and GPipe [47]. These techniques investigate splitting large DNN models across multiple machines for efficient computation. PipeDream partitions the DNN layers into different stages and executes them on the set of interconnected computing devices. It follows the async pipeline mechanism for which the process requires large memory to store the intermediate model parameters. This process pipelines the execution of forward passes and distributes them with backward passes asynchronously. GPipe, on the other hand, splits the minibatch of input data into multiple micro-batches. After that, it executes these micro-batches using a pipeline strategy across various devices. Finally, gradients are synchronously updated during backward passes. It holds a single-weight version and flushes some results to free the memory. However, this approach may suffer from increased computation overhead due to the need to recompute intermediate results for backward propagation. PipeDream-2BW presents a double-weight update mechanism to reduce the memory footprint but still experiences parameter staleness issues [92] and computation convergence dependent on the GPU platform.

While PipeDream, PipeDream-2BW, and GPipe have demonstrated improved results in terms of time-to-accuracy, it is important to mention that the architectures they propose are distinct from the architecture we introduce in our paper. The approaches above primarily target large-scale clusters with multiple GPUs or accelerators, whereas our architecture is designed for edge and embedded systems. Our primary focus is to explore the splitting of large DNN models in the context of our architecture to enhance their performance, specifically concerning model training time. Our method can be seen as complementary to the approaches presented in PipeDream-2BW, PipeDream, and GPipe, as it addresses a different aspect of optimizing large-scale DNNs. Combining our method with the techniques proposed in these studies may lead to further improvements in both model training time and time-to-accuracy by leveraging the strengths of each method within their respective architectures.

In study [93], the authors design an accelerator to reduce the computational requirements of CNNs by avoiding useless operations in sparse architectures. Their design aimed to improve performance and lower energy consumption in resource-limited embedded systems. The evaluation was carried out on Field-Programmable Gate Arrays (FPGAs) that support the sparsity of DNN models. Model pruning techniques, which are currently popular for data compression and reducing the number of Multiplication and Accumulation (MAC) operations by setting many parameters to zero, have been used in the field. However, this sparsity can lead to random memory access issues that can negatively impact performance in parallel architectures.

In some cases, the data parallelism technique does not offer much benefit, due to large deep learning model execution on resource-constraint devices. Therefore, model splitting is a potential option to achieve the parallel execution of large DNN models. The authors in [94] propose a layer partitioning approach that splits the neural network into multiple sub-networks, and distributes the partitions on edge devices for faster execution. To obtain the splitting point of a model, the proposed approach finds the areas with the largest differences in the consecutive layers at runtime. Moreover, the method maintains a model manager to make an early exit from training by calculating the model's confidence, which is then compared with an entropy threshold.

In another work [95], Li Zhou *et al.* propose a runtime adaptive CNN acceleration framework for deploying ML models on different devices. This approach suggests partitioning the workload into different devices, considering the devices' capabilities and network latency. In addition, it introduces two different strategies: layer-wise and fused-layer, for parallelizing CNNs in IoT devices. The layer-wise parallelization strategy executes a part of the model on edge devices where all the outputs of the sub-modules need to be merged before the subsequent output. This technique has a larger communication delay for distributed computation. On the other hand, the fused-layer strategy directly sends the result of one layer to the next layer, instead of merging the results before the next layer. The framework predicts the execution time of each layer, and the communication latency to select the best partition point. The experimental results offer a speed up between 1.9 to 3.7, using eight devices, for three CNN models (e.g., VGG-16 and YOLOv2).

Recent works on pipeline model parallelism show the need for model parallelism as large DNNs are not likely to fit in the local memory of a single machine [47], [96]–[98]. However, parallelizing them into different devices is still challenging, as the process introduces various performance issues, including implementation complexity. For example, PipeDream focuses on maximum throughput discarding the synchronous operations, which leads to large memory demand. This implementation will be a challenge considering the memory constraints of embedded systems. However, researchers have also investigated different memory-efficient model parallelism approaches without compromising the model performance. For example, PipeMare [98] and PipeDream-2BW [90] offer more memory-efficient approaches than PipeDream. In addition, DAPPLE [97] offers improved experimental results in terms of memory, speed up, and convergence compared to GPipe [47]. Regardless, finding the optimal splits based on available resources, which we present in this work, can be a potential solution for leveraging a parallel computing framework to improve embedded applications' performance.

Over the years, numerous task-scheduling algorithms have been studied to effectively allocate resources in both homogeneous and heterogeneous multiprocessing systems [99]. Wu *et al.* [100] propose a pipeline-based scheduler to optimize latency for CNN inference on heterogeneous multi-core systems, aiming to minimize feature-map data movement. The scheduler comprises two phases: the pipeline-configuration phase, which groups CPU cores into computing stages, and the layer-to-stage allocation phase, which maps segments of the CNN model to the pipeline stages. By employing this approach, the authors achieve a 73% performance improvement in throughput compared to existing multi-thread schedulers. With the growing demand for compute intensive deep learning applications, systems equipped with heterogeneous processors, such as multicore CPUs, GPUs, and NPUs, are becoming increasingly prevalent [101]. These applications often require concurrent execution of tasks with various constraints, leading to a growing interest in inter process communication (IPC) conscious scheduling to minimize the overall execution time through parallel implementations. In particular, cluster scheduling, which divides tasks and processing cores into multiple clusters, is becoming popular as it enables the migration of tasks within their groups [102].

While these studies have demonstrated improved results in terms of time-to-accuracy, our work's primary focus is to explore the partitioning of large DNN models to enhance their performance, specifically in terms of model training time on embedded devices. We addresses the design challenges of implementing pipeline model parallelization by determining optimal splits for a given DNN model and computing resources.

2.3 Summary

This chapter serves as a gateway into the intricate world of next-generation embedded software development, offering insights into the methodologies and challenges faced in feature identification, traceability, and the integration of advanced computational paradigms. The journey begins with exploring reverse engineering and its advantages in reusing existing applications for faster and less error-prone development. The role of static code analysis, particularly through tools like *cflow* and *pycparser*, emerges as a crucial component in understanding system design constraints and real-time requirements accurately.

The narrative then shifts to the landscape of software product lines, illustrating how they contribute to the efficient management and reuse of legacy software assets. This section underscores the importance of feature models in representing the variability and commonalities of application families, a concept crucial for managing change and evolution in software product lines. The chapter also investigates decentralized computing for embedded systems, emphasizing the need for distributed computing resources in processing massive amounts of sensor data. The discussion includes comparing wireless technologies that play a significant role in fog computing architecture, highlighting their applicability in different use cases. A significant portion of the chapter is devoted to parallel computing for machine learning in embedded systems. It explores various aspects, including data parallelism, model parallelism, and pipeline model parallelism, detailing their respective advantages and implementation challenges. The discussion extends to modern deep learning frameworks, emphasizing their role in supporting DNN model training and highlighting the unique challenges they address in the context of embedded systems.

In summarizing the related works, the chapter presents a comparative analysis of different software engineering methodologies used for feature identification and traceability. It highlights the advancements in topic modeling, NLP techniques for requirements extraction, and the various approaches used for feature location and extraction from source code. The comparison explains the diverse techniques and tools employed in different studies, offering a broad perspective on this field's state of the art. The chapter concludes by emphasizing the unique approach of our research in addressing the challenges inherent in CPS software reuse. By integrating static code analysis with topic modeling and leveraging advanced NLP techniques, our methodology aims to provide a comprehensive solution for identifying reusable features and requirements, thereby facilitating faster and more efficient development of next-generation embedded software.

Chapter 3

FEATURE-BASED REUSABLE EMBEDDED SOFTWARE DESIGN

3.1 Introduction

Embedded systems and Cyber-Physical Systems (CPS) are at the forefront of technological advancements, playing pivotal roles in various sectors, including healthcare, automotive, aerospace, and smart cities [103], [104]. By integrating computational processes with physical systems, embedded systems offers intelligent functionalities and interacts dynamically with its environment. Despite the promising potential, the development of embedded software is filled with complexities. Ensuring safety, performance, and security remains a significant challenge, especially under unpredictable operating conditions. This complexity is heightened by the multidisciplinary nature of embedded systems, demanding a mix of expertise, rigorous testing, and quality assurance measures.

An effective strategy to alleviate these challenges lies in software reuse. Reusing software assets across different projects offers advantages such as reduced development time, cost savings, and minimized risks associated with creating new modules. However, the adoption of software reuse in embedded systems, is limited due to the difficulties in identifying reusable features from legacy repositories [3], [105]. Often, these repositories, like GitHub, lack comprehensive documentation, making manual feature extraction tedious and error-prone.

Recent automation techniques, utilizing code analysis combined with machine learning, present promising solutions to address these challenges [106], [107]. Specifically, topic modeling techniques such as Latent Dirichlet Allocation (LDA) can filter latent topics from large text collections, potentially identifying features and requirements [108]. When paired with cutting-edge deep learning models like the Bidirectional Encoder Representations from Transformers (BERT), a sound understanding of software behavior can be achieved, paving the way for effective software reuse [32].

In light of these considerations, this chapter introduces an innovative automated methodology that combines static code analysis, BERT's semantic understanding, and topic modeling. The central aim is to revolutionize the software reuse process, thereby optimizing embedded software design, development, and overall efficiency. The key contributions of this chapter enclose:

- The introduction of a holistic methodology that combines static code analysis, BERT-based semantic analysis, and topic modeling with the innovative use of call graphs to extract feature and Requirements in embedded software.
- The development of an embedded software feature model that presents relationships among reusable functions, supporting the feature-to-artifact mapping and enhancing the understanding of software requirements.
- A thorough evaluation of the introduced methodologies against existing manual and rule-based methods, highlighting their real-world robustness and potential to streamline embedded software development.

3.2 System Architecture for Next-Gen Embedded Design

Our proposed architecture for automated identification of features and requirements in embedded software is constructed through a series of stages. These include static analysis, preprocessing, topic modeling, BERT-based deep semantic analysis, and the strategic mapping of identified features to their related functional and non-functional requirements. A visual representation of this architecture can be found in Fig. 3.1.

Artifact: An artifact refers to a distinct and self-contained segment of source code, often representing a functional or structural unit, which can be analyzed to extract particular features or attributes. This can encompass functions, classes, modules, or even entire applications, depending on the granularity of the analysis.

Static Analysis Module

This stage ensures the systematic breakdown of input sources, enabling the extraction of essential artifacts and preparing the entire codebase for deeper analysis. As an example, it extracts code artifacts, such as the longest common units from the function call graph that is generated using the *cflow*. It reveals software

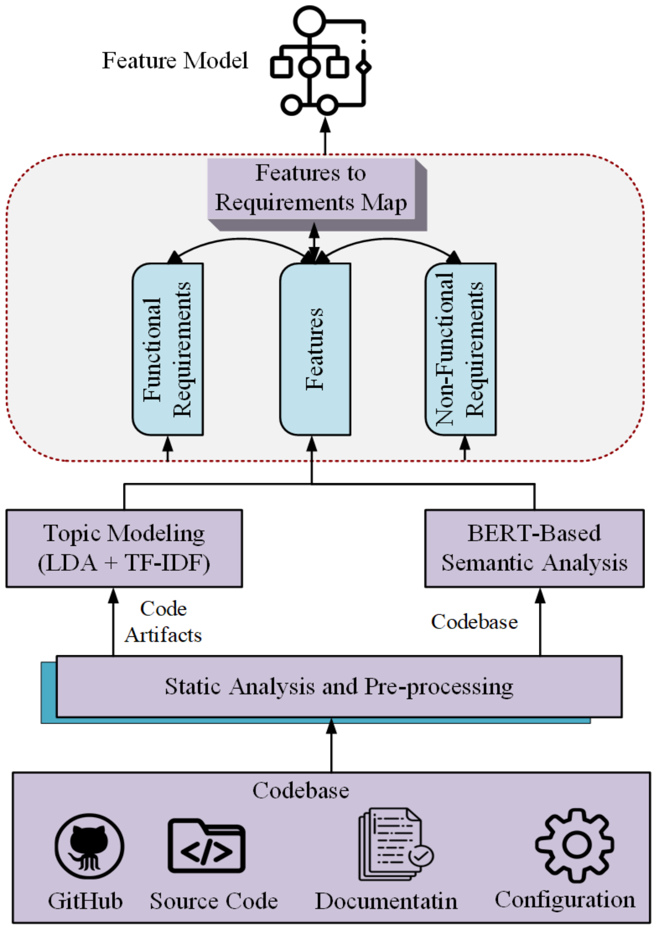

Feature Model
Features to Requirements Map
Functional Requirements
Features
Non-Functional Requirements
Topic Modeling (LDA + TF-IDF)
BERT-Based Semantic Analysis
Code Artifacts
Codebase
Static Analysis and Pre-processing
Codebase
GitHub
Source Code
Documentatin
Configuration

Figure 3.1: System architecture for next-generation embedded software design

functions, their interactions, and dependencies. For a given code segment, represented by X, the function call graph can be defined as $G(X) = \phi_1(X), \phi_2(X), ..., \phi_n(X)$ where each $\phi_i(X)$ denotes a specific function and its interactions within the code snippet X.

Topic Modeling Module

After preprocessing, the topic modeling module steps in to mine the codebase's structure. Using different methods such as TF-IDF and LDA, this module discerns topic-based insights to extract both features and requirements.

BERT-Based Semantic Analysis Module

This module leverages the fine-tuned BERT model to parse the source code and associated documentation, especially focusing on identifying non-functional requirements and providing a deeper semantic understanding of potential features. The results from BERT and the topic modeling module are then integrated to form a refined list of features and requirements. To formalize this, the features and requirements of a code segment, denoted as X, are represented as vectors $F(X) = f_1, f_2, ..., f_k$ and $R(X) = \oplus r_{f1}, \oplus r_{f2}, \otimes r_{nf1}, \otimes r_{nf2}, ...$, respectively. The $\oplus$ symbol represents functional requirements, while the $\otimes$ symbol represents non-functional requirements.

Features to Requirements Mapping Module

Informed by insights from both static analysis and the combined BERT and topic modeling results, this module systematically assigns features to their corresponding requirements. Requirements are categorized into functional (R_f) and non-functional (R_{nf}) types. The mapping from features to functional requirements is represented as $M(F) = R_{f1}, R_{f2}, ..., R_{fn}$, and to non-functional requirements as $M'(F) = R_{nf1}, R_{nf2}, ..., R_{nfm}$. These mappings explain the relationships between particular features and their related requirements, promoting effective software reuse.

Feature Model

The insights gained from the static code analysis, BERT-based analysis, and topic modeling modules are integrated to create a comprehensive set of features and requirements. This data is then used to construct a feature model that can facilitate software reuse in embedded system development. This inventory subsequently informs the design of a feature model, detailing relationships (mandatory, optional, OR, XOR, Requires, Excludes) between features, thus supporting software reuse in CPS particularly in embedded system development.

Assumptions

The following assumptions are made in the proposed system model:

- The source code is sufficiently complex and rich in semantics to enable the extraction of meaningful features and requirements.

- The combined prowess of topic modeling and BERT-based analysis can effectively identify latent topics and semantic intricacies resonating with features and requirements.

3.3 Methodologies for Legacy Code Reuse

This section explains the methodologies for facilitating the reuse of legacy embedded software, detailing a comprehensive approach that evolves from foundational static code analysis to incorporating advanced techniques like topic modeling and BERT embeddings. Our journey begins with the basic principles of reusable function identification utilizing static analysis, laying the groundwork for understanding and extracting key features from legacy code. We then progress to more advanced methods, enhancing our ability to determine features and requirements integral to modern embedded systems. The proposed approach combines established practices and innovative strategies to construct the feature model.

3.3.1 Base Approach for Reusable Function Identification and Feature Model Construction

This base proposed approach finds reusable functions and presents them in a feature model using the following steps:

- Identifying the reusable functions: It identifies the reusable functions from legacy embedded software code.
- Identifying the function relationships: It determines the function type and dependencies.
- Feature model construction: A new level-based feature model construction algorithm is described to map all the reusable functions as features in the feature model.

Listing 3.1: C Code 1

```c
void navigate(int rotate) {
    set_direction (rotate);
   fly_normal ();
}
void takeoff(float sp, float alt) {
   if (sp>0){
    navigate(rotate);
   }
   gps_tracker ();
}
void landing(float distance) {
    safe_return (distance);
   gps_tracker ();
}
 float  cal_altitude () {
   return  gps_tracker ();
}
void controller (int mode) {
   switch(mode) {
   case 1:
         takeoff ();
         break;
   case 2:
         landing ();
         break;
    default :
        fly_normal ();
         break;
   }
}
```

Listing 3.2: C Code 2

```c
void navigate(int rotate) {
   if (rotate != NULL)
         set_direction (rotate);
   else
        fly_normal ();
}
/*sp = speed, alt = altitude */
void takeoff(float sp, float alt) {
   if (sp>0){
    navigate(rotate);
   }
   gps_tracker ();
}
void landing(float distance) {
    safe_return (distance);
   gps_tracker ();
}
 float  gps_tracker () {
   return  get_coordinate ();
}
 float  get_coordinate () {}
int main(){
     while(get_connect){
        /*use controller to change mode*/
         controller (mode);
     }
     landing (); /*upon battery charge */
}
```

Identifying the Reusable Functions

In the context of identifying reusable functions in embedded software, a function call graph is generated from a given legacy embedded software using GNU's *cflow* graph generator [109]. This control flow graph captures the invocation relationships and subroutine information of the program. Subsequently, various attributes, such as the number of functions, the signature of each function, the function's definition, its location, and the depth of function calls, are retrieved by modifying the *pycparser* python library [110]. Due to the potential for long sequences of function calls, including recursive function calls, flow graphs can be expansive. Therefore, to handle the recursive function as a typical function call, we place an extra 'R' symbol beside the function's name.

- Reusable function identification: Functions are identified as reusable if they appear more than a specified threshold number within a program. An optional threshold value is set, allowing developers to modify this value and limit the function list for visualization at different abstraction levels. The process:
  - Retrieves the list of functions, encompassing their parameters, variables, and constants. A function is mapped in the standard form of $f_i \rightarrow f_j$, correlating directly with the source code.
  - Identifies constraints by analyzing various conditional statements, such as if-else, while loops, and switch conditions, from both the code and developers' comments.

Two C code samples are provided, as seen in Listing 3.1 and 3.2, to explain how highly reused functions are considered as potential features. In these examples, the assumption is made that any function called more than once is selected a potential reusable feature. Therefore, in this context, functions like $navigate()$ and $gps\_tracker()$ are identified as features due to their multiple invocations. Additionally, constraints related to functions like $takeoff()$ and $landing()$ are also extracted. Variables such as sp, $mode$, and $rotate$ are implicated in determining certain constraints through their utilization in if-else or switch statements for calling other sub-features. Only user-defined functions are considered, while library functions are excluded based on their function definitions, streamlining the approach. If no threshold value is set by developers, by default, all available functions are retrieved as features.

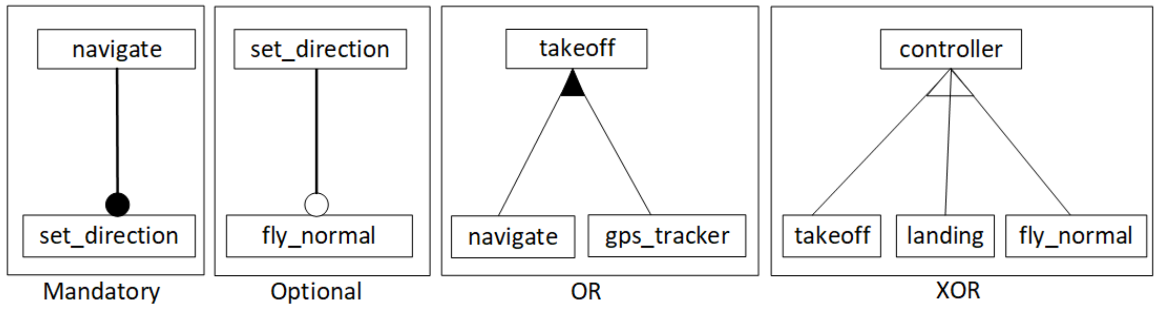


Figure 3.2: Feature types and relationships

Identifying the Function Relationships

In this step, the proposed approach defines different types of functions and their relationships. The process transforms the corresponding functions into features in the feature model of an embedded software. We demonstrate the relationships in Figure 3.2 for better visualization.

- *Mandatory*: A function is called a *mandatory* feature if and only if its parent node requires to execute it all the time. All its mandatory child features must also be included (n from n) in a parent feature. As an example, $set\_direction()$ and $fly\_normal()$ is a mandatory feature for $navigate()$ in Listing 3.1 where $gps\_tracker()$ and $safe\_return()$ are two mandatory sub-features under $landing()$.
- *Optional*: A function is an *optional* feature if it is selected with other available functions. It means an optional feature can frequently appear along with a particular feature, but it may appear with other features too. Any number of features can be added (m from n, $0 \leq m \leq n$) under a parent feature. For example, $fly\_normal()$ is an optional feature to $set\_direction()$ which is shown in Figure 3.2.
- *Alternative (XOR)*: A function is in *alternative (XOR)* feature group if it only gets selected among other functions based on certain conditions. In the case of *alternative/XOR* relationship, exactly one feature must be selected from a group of alternative features (1 from n). Thus, features $takeoff()$, $landing()$, and fly_normal of Listing 3.1 are in *XOR* relationship (because of the switch statement).
- *OR*: A function is in *OR* relationship if its parent function selects at least one function out of multiple functions (m from n, m>1). Thus, $navigate()$ and $gps\_tracker()$ are in *OR* relationship in $takeoff()$.
- *Exclude*: If a particular function is selected from a group, other functions from the same group cannot

be selected. In this relation, one function eliminates another function for selection.

- *Requires*: A *requires* relation indicates the dependency of one function (source) to another (target) function.

Moreover, we reorder the function relationships to make them more manageable for adapting to a feature model. For example, Listing 3.1 and 3.2 show that $navigate()$ is a reusable feature and it is connected with other mandatory (e.g., $set\_direction()$) and optional features (e.g., $fly\_normal()$). Therefore, the partial feature model is as follow:

$$F' = (navigate() \leftarrow set\_direction() \wedge fly\_normal()) \vee$$
$$(navigate() \leftarrow set\_direction() \wedge \neg fly\_normal())$$

Let us consider a case where $fly\_normal()$ is considered as optional feature. In Listing 3.1, it is shown that $fly\_normal()$ appears with a mandatory feature $set\_direction()$ but with XOR relation in Listing 3.2. Thus, we define $fly\_normal()$ as an optional sub feature of $set\_direction()$.

While both $navigate()$ and $takeoff()$ features show the possibility of reuse as combined or in an alternative to each other, we define them with a *OR* relationship. To define a *OR* relation, we assign the features under its parent feature, which is shown in the following partial model.

$$F' = takeoff() \leftarrow (navigate() \vee \neg navigate()) \wedge gps\_tracker())$$

Furthermore, the dependencies among the features are simplified in tree-structured child-parent relationships. As an example, Figure 3.3(i) shows that feature $main()$ has two mandatory features $controller()$ and $landing()$ where feature $landing()$ is also a mandatory feature of $controller()$. Therefore, we define the tree as $main() \rightarrow controller() \rightarrow landing()$.

However, a combination of multiple independent features can form another new feature. We define them as sub-features of the new feature in a sub-tree. In such a case, we assign the relation of that independent feature with the new feature using *Requires*. In Figure 3.3(ii), $cal\_altitude()$ is identified as an independent feature of a sub-tree where feature $gps\_tracker()$ is a part of the main tree. When the feature $cal\_altitude()$ calls $gps\_tracker()$, it gets connected with *Requires* relationship.

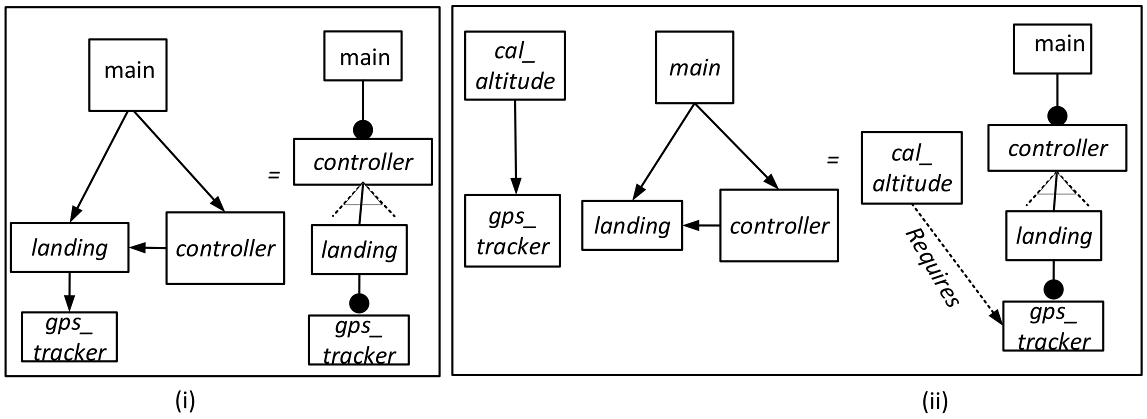


Figure 3.3: Simplification of function relationships for feature model.

Feature Model Construction using a Level-based Algorithm

To construct the feature model, we propose a level-based algorithm that shows how the features are placed into different levels and what information is required for implementing a particular application. We implement a python tool for automating the proposed level-based feature model construction algorithm where a list of features is distributed with their associate feature types and constraints. At the beginning of this process, we determine the level of each feature using the call graph. Thus, features are mapped at each level based on the number of incoming function calls in the program. The hierarchy of function calls specifies the parent and child relationship. This relationship is identified from the caller and callee function list of the call graph. To find out the number of incoming function calls of each feature, we define a matrix named F^{in}. As an example, the matrix F^{in} represents a n-by-n matrix where n is the number of features. The degree of incoming function calls of each feature from every other feature is denoted by d_{ij}. It stores the corresponding degree of incoming function calls to feature f_i from f_j. To simplify the feature model, we set the number of incoming calls to equal one for any recursive function.

$$F^{in} = \begin{array}{c|c} & \text{Features} \longrightarrow \\ \text{Features} \downarrow & \begin{array}{c|ccccc} & 1 & 2 & 3 & .. & n \\ \hline 1 & d_{11} & d_{12} & d_{13} & .. & d_{1n} \\ 2 & d_{21} & d_{22} & d_{23} & .. & d_{2n} \\ 3 & d_{31} & d_{32} & d_{33} & .. & d_{3n} \\ .. & .. & .. & .. & .. & .. \\ n & d_{n1} & d_{n2} & d_{n3} & .. & d_{nn} \end{array} \end{array}$$

After that, we count the total incoming calls of each feature, adding all the corresponding row values using Equation 3.1.

$$f_i^{in} = \sum_{j=1}^{n} d_{ij}; where\ d_{ij} = \begin{cases} x, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

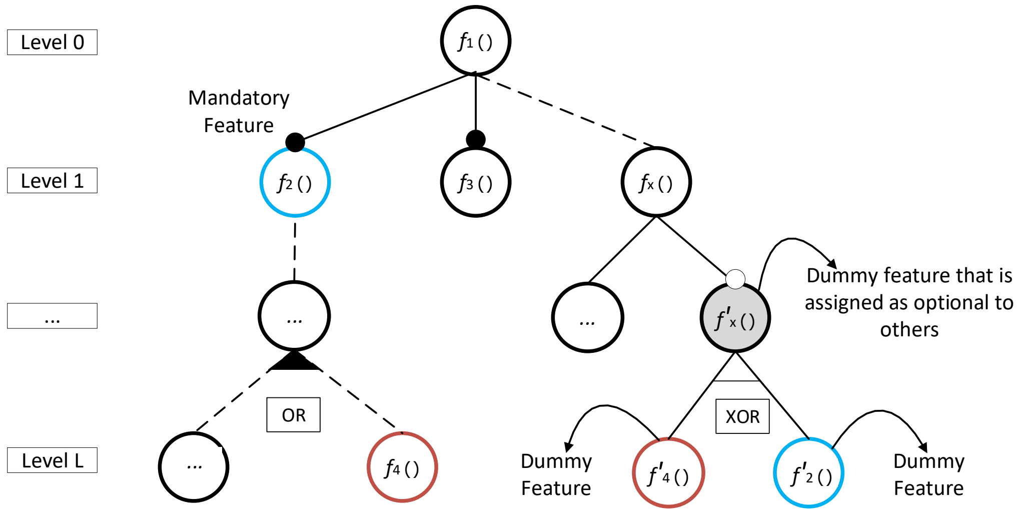


Figure 3.4: An implementation concept of feature model construction.

Finally, we use the following steps to construct the feature model in a tree structure.

- First, we assign the function as a root feature that has a minimum incoming or maximum outgoing function calls in the program and does not have any parent.
- Second, we place all the features at different levels according to the number of incoming calls. Therefore, the root feature is placed at level 0, intermediate features are between level 0 to L, and the leaf

Algorithm 1: Find the Levels of Reusable Functions and Their Parent-Child List

```
Input   : Source code of embedded software
Output: Features levels including parent-child information
/* functions of embedded software */
$featureList$ []= Extract functions from call graph
$G$ = nx.DiGraph() /* call graph */
$fm$ = nx.DiGraph() /* feature model */
/* finds the levels of features */
$level\_list$[][] = List of features at each level
for each feature $i$ in range(len($featureList$)) do
    s=featureList[i];
    /* finds the level of a feature */
    k=int(re.search(r$'\setminus d+'$, s).group())
    $level\_list$.append(k)
end
/* finds the parents of features */
$pr$ = [[] for x in range(len($featureList$))]
for each feature $i$ in range(len($featureList$)) do
    /* check the parent of a feature */
    nd = [x for x,y in $G$.nodes(data=True) if y[$'value'$]==$featureList$[i]];
    for $j$ in range(len(nd)) do
        for $k$ in $G$.predecessors(nd[j]) do
            pr[i].append($G$.nodes[k][$'value'$])
        end
    end
end
/* finds the child of features */
$ch$ = [[] for x in range(len($featureList$))]
for each feature $i$ in range(len($featureList$)) do
    nd = [x for x,y in $G$.nodes(data=True) if y[$'value'$]==$featureList$[i];
    for $j$ in range(len(nd)) do
        for $k$ in $G$.successors(nd[j]) do
            ch[i].append($G$.nodes[k][$'value'$])
        end
    end
end
buildFeatureModel($featureList$, $level\_list$, $ch$, $pr$);
```

Algorithm 2: Feature Model Construction

```
Input  : $featureList$ [], $level\_list$[][], $ch$[], $pr$[], $G$, $fm$
Output: Feature model
Function buildFeatureModel(Input)
    for each level l in range(len(level_list)) do
        for each feature $i$ in level $l$ do
            $G$.add_node(i,value=$level\_list$[l][i])
            if level $(l-1)$ finds child in list $ch$ then
                $fm$.add_node($level\_list$[l][i])
                $fm$.add_edge(i,$level\_list$[l-1][i])
            else
                $fm$.add_node(i,$level\_list$[l][i])
            end
        end
    end
    for i in $fm$.edges.data() do
        /* return relation type */
        k=find_relation(i[0],i[1])
        /* return constraints if any */
        c=find_constraint(i[0],i[1])
        $fm$.edges[i[0],i[1]][$'relation'$]=k
        $fm$.edges[i[0],i[1]][$'constraint'$]=c
    end
    /* display feature graph */
    drawNetwork($fm$);
return
```

features with maximum incoming calls are placed into the last level L.

- Third, if any child feature is mandatory and located at a different level than the subsequent level of its parent, we can not include it in an *alternative (XOR)* relationship. We set the feature as a *dummy* feature for *XOR* relation under its parent feature. In Figure 3.4, we present an example for creating a *dummy* feature $f_2^{'}()$ that replicates the mandatory feature $f_2()$ at Level L.
- Fourth, if any feature is in *OR* relationship, we can not put it under the *XOR* relationship in a different place. To combine it with the *XOR* relationship, we define it as a *dummy* feature that replicates the original feature. The created *dummy* feature $f_4^{'}()$ with an *XOR* relation is shown in Figure 3.4.
- Fifth, if we create a *dummy* feature to build an *OR/XOR* relationship, we introduce their parent as *dummy* and make it as an *optional* feature to other features.

During the feature model construction, we traverse all the features from the lowest level L to the highest level 0. The distinct features of the lowest levels are considered leaf nodes in the feature model. We gradually decrease the level value to find the corresponding features mapped with its child features (callee function). The relation of each child feature is characterized by considering the feature types. Moreover, the *requires* and *excludes* relationships are identified from the sub-feature model relation. A feature is connected with *requires* relation if the feature is a *mandatory* feature at any level but also appears as a mandatory to a different level of another feature. On the other hand, the *excludes* relation identifies the feature that never appears with a specific feature that likely appears with all the common features.

The overall process of the level-based feature modeling is demonstrated in Algorithm 1 and 2. Algorithm 1 extracts the reusable functions as features through the call graph and then identifies each feature's level from the call graph. The parent and child feature list is classified considering the caller and callee relationship. After that, we add the feature relation and constraints to build the valid configurations in the feature model. Finally, Algorithm 2 builds the feature model in a tree format.

In summary, the above base methodologies form the foundation of our approach to feature identification. By establishing a solid foundation with function call graphs and level-based feature modeling, we create a robust framework for understanding the intricate relationships within legacy software. This groundwork leads the way for integrating more advanced techniques, setting the stage for a more in-depth exploration of feature extraction.

3.3.2 Enhancing Feature and Requirements Identification using Topic Modeling and BERT

Building upon the foundational techniques of reusable function identification and feature model construction, we now focus on further enhancing the feature and requirements identification process. Recognizing the need to avoid the pitfalls of over-granularity in function-based features, we introduce advanced methodologies such as Topic Modeling and BERT. These methods extend our analysis and extraction of features and requirements from embedded software by moving beyond structural relationships to find semantic interconnections within the software's codebase. This subsection explores how Topic Modeling and BERT facilitate a more refined and comprehensive understanding of software features and requirements. It ensures a robust analysis, enabling a more holistic and effective identification of important features. A comprehensive exploration of each phase can be found in the ensuing subsections, while visualization of the methodology is captured in Algorithm 3.

- Preprocessing and Functional Properties Identification
- Feature Discovery using LDA and BERT
- BERT-Enhanced Requirement Extraction

Preprocessing and Functional Properties Identification

To analyze the source code, we first investigate its functional characteristics and then prepare it for further examination (preprocessing).

- **Functional Properties Identification:**
  - Function Call Graph: The function call graph, denoted as $G = (V, E)$, displays how functions in the software interact. Here:
  $$G = \{V, E\}$$
  With:
    - V being the set of all functions.
    - E representing the interactions between these functions.
  - Function Relations and Dependencies: We express relationships between functions using a matrix D. Each element d_{ij} shows the type of relation between function v_i and v_j:
  $$D = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{bmatrix}$$
  - Function Group Identification: Functions are grouped based on their role. We denote functions as F, and each function ϕ is assigned to a group $G(\phi)$:
  $$G : \phi \to \{\text{Core, Auxiliary, Dependent, Exclusive}\}$$
  Where:

    * **Core** are functions that are fundamental to the software's operation.
    * **Auxiliary** functions supplement or enhance the Core functions.
    * **Dependent** functions require one or more Core or Auxiliary functions.
    * **Exclusive** functions cannot co-exist with certain other functions.

- **Preprocessing:**
  - Artifact Extraction: We transform the source code into a series of artifacts. Depending on the programming paradigm and software architecture, these artifacts may represent functional units, object-oriented modules, classes, or other code segments. Let A be an artifact, representing a segment of source code. The granularity of A can range as:

$$A \in \{\text{function, module, class, application}\}$$

  - Tokenization: Each component A is further divided into smaller units called tokens, represented by the set T:

$$A \to T = \{t_1, t_2, ..., t_n\}$$

  - Cleaning and Refinement: We remove non-essential tokens, represented as T', to get a refined set T_{clean}:

$$T' = \{\text{whitespace, delimiter, ...}\}$$

$$T_{clean} = T - T'$$

  - Pattern Analysis: The Longest Common Unit (LCU) technique helps identify redundancies or shared functionalities and patterns among shared artifacts. For a set of source code artifacts $A_1, A_2, ..., A_m$, the LCU is defined as:

$$\text{LCU}(A_i, A_j) = \max_{k,l}\{t_k \in A_i, t_l \in A_j | t_k = t_l\}$$

Algorithm 3: Features and Requirements Identification from Embedded Software

```
Input   : Collection of source code files $\mathcal{C}$
Output: Identified features, requirements, and their mappings $\mathcal{F}$
/* Static Code Analysis */
1  $T \leftarrow$ TokenizeSourceCode($\mathcal{C}$)
2  corpus $\leftarrow$ BuildCorpus($T$)
/* Topic Modeling using LDA */
3  ldaModel $\leftarrow$ TrainLdaModel(corpus)
4  topics $\leftarrow$ ExtractTopics(ldaModel)
/* BERT-enhanced Feature and Requirement Extraction */
5  Function Identification(topics, T)
       /* Provide LDA topics and tokens to BERT */
6      $bi \leftarrow$ CombineTopicsAndTokens(topics, $T$)
       /* Feature Extraction using BERT */
7      $F(X) \leftarrow$ BERTExtractFunctionNames($bi$)
       /* Requirement Extraction using BERT */
8      $R_f(X) \leftarrow$ BERTExtractConditions($bi$)
9      $R_{nf}(X) \leftarrow$ BERTExtractComments($bi$)
10     $R(X) \leftarrow$ Combine($\oplus R_f(X)$, $\otimes R_{nf}(X)$)
       /* Map Requirements to Features */
11     $f\_map \leftarrow$ BERTSimilarity($F(X)$, $R(X)$)
12     $\mathcal{F} \leftarrow F(X) \cup \oplus R_f(X) \cup \otimes R_{nf}(X) \cup f\_map$
13     return $\mathcal{F}$
```

Feature Discovery using LDA and BERT

Upon the completion of preprocessing and identifying functional properties, the next step involves the exploration of potential features within the source code artifacts. To determine the features, we focus on dominating functionalities and characteristics of the software, avoiding granular and low-level code details that might not represent a meaningful feature in the context of embedded systems. This is crucial to ensure that the discovered features are significant, actionable, and not lost amidst the minute code intricacies.

This section presents the methods and models utilized for this purpose, blending conventional and advanced techniques to extract relevant software features and requirements. Starting with the basic representation, for any artifact A, its Bag-of-Words vector is denoted as V:

$$V = \{f(t_1), f(t_2), ..., f(t_n)\}$$

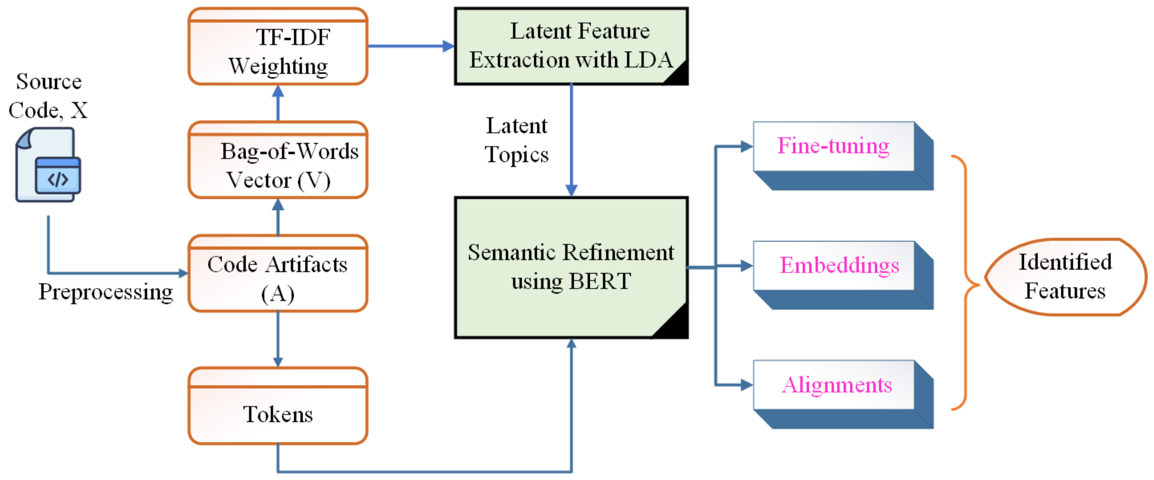


Figure 3.5: Workflow of feature identification, transitioning from source code artifacts to potential feature extraction

Here, $f(t_i)$ indicates the occurrence count of token t_i in A. Additionally, to stress the significance of each token across the entire dataset, we utilize the TF-IDF weighting:

$$\text{TF-IDF}(t_i, A) = f(t_i, A) \times \log\left(\frac{|A|}{1 + \text{count}(t_i, A)}\right)$$

- **Latent Feature Extraction with LDA**

  LDA is then applied to find out hidden topics or themes within the data. These latent topics often resonate with potential software features or requirements. This phase of topic extraction can be visualized in Figure 3.5. In our approach, the number of topics for LDA was decided based on iterative testing and evaluation, aiming to balance granularity and coherence. We experimented with a range of values and observed the topics' relevance and distinctiveness, settling on a number that offered our data's most meaningful and distinct topics. Additionally, we acknowledge the existence of alternative algorithms to LDA, such as Non-Negative Matrix Factorization (NMF) and Hierarchical Dirichlet Process (HDP), which could offer different perspectives in topic modeling. Future work could explore the integration of these methods to enhance our feature extraction process.

- **Semantic Refinement using BERT**

  The CodeBERT [111] model, known for its prowess in understanding code context, is tailored for our dataset:

1. Initially, BERT is fine-tuned to capture the domain-specific semantics.
2. The topics classified by LDA and codebase are added to BERT's embeddings, offering a richer semantic understanding.
3. By aligning the classifications from BERT with the LDA topics, we ensure a comprehensive and semantically accurate identification of features.

BERT-enhanced Requirements Extraction

In the process of software requirement extraction, differentiation between functional and non-functional requirements is important. While BERT's capabilities enrich the semantic extraction process, additional methods and heuristics are necessary to grasp the intricacies of both requirement types.

- Functional Requirements Extraction:

  Functional requirements describe what the system should do, often encapsulated in the logic and conditions of the code.

  - Extracting Conditional Constructs: As an initial heuristic, conditions in `if`, `while`, and `for` loops provide insight into system behaviors and rules.
  - Function and Method Analysis: Function names and method signatures often describe actions or operations, indicating the functionalities the software provides.

- Non-functional Requirements Extraction:

  Non-functional requirements capture system properties or characteristics rather than behaviors. Here's a more nuanced approach:

  - Extracting Comments: While comments are a starting point, not all comments correspond to non-functional requirements. Comments that capture constraints, quality attributes, or overarching goals are prioritized.
  - Documentation Analysis: Often, non-functional requirements like performance, scalability, and security are better documented in READMEs, architecture documents, or inline documentation blocks.

- Code Patterns and Libraries: Certain code patterns or imported libraries may hint at non-functional requirements. For instance, importing a security library might indicate security concerns.

To verify that the requirements are satisfied by the identified features, we cross-referenced the extracted features and requirements with the manually established ground truth. Additionally, we compute the cosine similarity to evaluate the alignment between the identified features and requirements extracted using the BERT model. This step ensures that the requirements identified are present in the code and functionally relevant.

Referring to Listing 3.3, the functional requirements of the electric water heater system are predominantly derived from its core functionalities. Features such as `initializeHeater` and `fillTank` clearly describe specific operations the system is expected to perform. For example, conditions like `user_pref == AUTO` and `!safetyCheck()` describe the system's behavior under certain situations, illustrating functional requirements. On the other hand, comments and annotations like *User Mode Selection: Manual or Auto* serve as a reflection of the non-functional requirements.

The overall feature and requirement identification process is detailed in Algorithm 3. Initially, source code files undergo static analysis, followed by tokenization, setting the stage for topic extraction through Latent Dirichlet Allocation (LDA). The hyperparameters for the LDA algorithm, including the number of topics, were set based on an initial fixed choice of 10 and the first top topic (features) is then extracted from the topics of each group. Leveraging the LDA topics and tokens, a BERT-enhanced method pinpoints software features and differentiates between functional and non-functional requirements. Lastly, BERT's capabilities map the discovered features to the relevant requirements, offering a concise overview of the software's intrinsic structure and objectives.

3.3.3 Feature Model Construction

Feature modeling presents a systematic approach to portray the variability and interdependence within software systems. The function frequency, and LCU analysis, enriched by TF-IDF and LDA (as previously discussed), form the basic layer of feature extraction. By incorporating BERT, we augment this base by accounting for semantic nuances embedded in code comments and related documentation, allowing for a more enriched and precise feature model construction.

Listing 3.3: Electric Water Heater Code

```
void initializeHeater () {
    temperatureControl (); // Core feature, hence Mandatory
}
// User Mode Selection: Manual or Auto
if(user_pref == MANUAL) {
    setTemperatureManually(); // XOR with autoTemperatureSetting due to the mutually
        exclusive if-else condition.
} else if (user_pref == AUTO) {
    autoTemperatureSetting (); // XOR with setTemperatureManually.
} else {
    defaultTemperatureSetting (); // Optional
}
// Dependent functions "Requires" relationship.
bool fillTank () {
    if (!safetyCheck()) { // The fillTank operation depends on the safetyCheck
        return false;
    }
    // Fill tank logic
    return true;
}
bool safetyCheck() {
    // Check safety conditions
    return conditionsSafe;
}
//"Excludes" relationship.
void activateEcoMode() { // Activates energy-saving mode
    deactivateTurboMode();
}
void deactivateTurboMode() { // Represents an "Excludes" relationship with activateEcoMode.}
// "OR" relationship.
for(int i = 0; i < MAX_POWER_LEVELS; i++) {
    if(isPowerLevelSuitable(i)) {
        setPowerLevel(i);
        break;
    }
}
```

Feature Relationships

BERT's prowess in understanding contextual nuances in textual data plays a pivotal role in discerning intricate relationships between software features. These relationships include:

- Mandatory: A feature that must be included whenever its parent feature is selected.
- Optional: A feature that can be chosen at discretion when its parent feature is included.
- OR: One or more among the child features must be selected if the parent is included.
- XOR (Alternative): Exactly one among the child features can be chosen when the parent is selected.
- Requires: The presence of one feature necessitates the presence of another.
- Excludes: The presence of one feature prohibits the inclusion of another.

Mapping Function Categories to Features

With BERT's enhanced capability, the semantic context surrounding functions becomes clearer. This clarity is leveraged to augment the categorization of functions into Core, Auxiliary, Dependent, or Exclusive, forming a robust foundation for our feature model.

Code Analysis

Consider the given code snippet in Listing 3.3 of the Electric Water Heater System:

From the above code Listing 3.3:

- Mandatory: Functions like *initializeHeater()* serve as the foundation of the system.
- XOR: The 'if-elseif-else' structure used for temperature settings denotes an XOR relationship.
- Optional: Functions like *defaultTemperatureSetting()* act as optional features.
- Requires: The direct function call within *fillTank()* denotes a 'Requires' relationship with *safety-Check()*.
- Excludes: The mutual inhibition between *activateEcoMode()* and *deactivateTurboMode()* shows an 'Excludes' relationship.
- OR: The loop iterating through power levels signifies an OR relationship.

The integration of BERT within the system promises to further refine these classifications by analyzing the semantics surrounding the code constructs and their associated comments. This could, for instance, highlight deeper interdependencies or requirements not immediately evident through structural analysis alone.

Theorem 1. Let $\mathcal{F}$ be a set of features. A feature model $\mathcal{M}$ accurately represents the relationships among features in $\mathcal{F}$ if and only if, for each feature $f \in \mathcal{F}$, $\mathcal{M}$ satisfies the constraints set by these relationships.

Proof. To explain, let the relationships be denoted as: mandatory (M), optional (O), OR (R), XOR (X), requires (Q), and excludes (E).

Consider a particular feature $f \in \mathcal{F}$. The very nature of relationships involving f dictates how f integrates with other features in $\mathcal{F}$. As an illustration, if f is mandatorily related to another feature g, it signifies that the selection of g mandates the selection of f as well.

($\Rightarrow$) Commence with the assumption that $\mathcal{M}$ captures the entirety of relationships in $\mathcal{F}$. By this virtue, for every feature $f \in \mathcal{F}$, $\mathcal{M}$ inherently satisfies the constraints (M, O, R, X, Q, E) involving f.

($\Leftarrow$) On the flip side, presume that for each feature $f \in \mathcal{F}$, $\mathcal{M}$ upholds the constraints (M, O, R, X, Q, E) associated with f. Relying on the definitive nature of these relationships, one can infer that $\mathcal{M}$ encapsulates the entire fabric of relationships amongst features in $\mathcal{F}$.

Feature Model Visualization

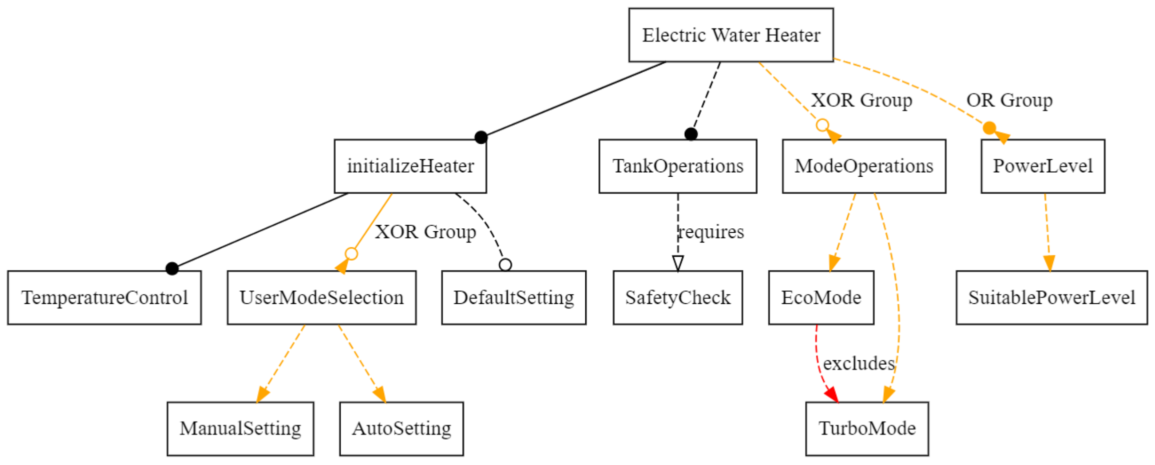


Figure 3.6: Feature model for electric water heater

The feature model for the Electric Water Heater (EWH), as visualized in Figure 3.6, presents a structured hierarchy that encapsulates the relationships, interdependencies, and intricacies of the functionalities. This structure is founded upon the core node, labeled `Electric Water Heater`.

Through BERT's semantic analysis of the associated comments and documentation, the model gains further precision. For example, the system's emphasis on user experience is clarified, highlighting the significance of user mode selection. This aids in understanding why, within `Initialization`, the XOR group `UserModeSelection` exists, offering alternatives like `ManualSetting` and `AutoSetting`. Similarly, a semantic analysis may highlight the prominence of energy efficiency concerns within the system, offering insights into why modes like `EcoMode` and `TurboMode` are present and mutually exclusive.

3.4 Evaluation of Automated Feature and Requirements Extraction

To validate the effectiveness of our approach, we deployed a series of experiments using real-world codebases from various domains, particularly Electric Water Heaters [112] and Microwaves [113], followed by the Autonomous Vehicle Systems (AVS) as a comparative analysis. This multi-steps approach ensured a comprehensive evaluation of our methodology. To establish ground truth for our evaluation, we manually analyzed the GitHub projects, identifying and listing features and requirements present in the code. This manual process involved a detailed source code inspection, comments, and available documentation. The ground truth thus established served as a benchmark for comparing the results of our automated feature and requirement extraction methodology.

Table 3.1: Feature and requirement identification results

Metrics	**Electric Water Heater**		**Microwave**	
	LDA based TF-IDF	**BERT**	**LDA based TF-IDF**	**BERT**
Total number of Functions	129		53	
Features Identified	17	29	16	18
Functional Requirements	63		61	
Non-Functional Requirements	10		4	

3.4.1 Feature and Requirement Identification

Table 3.1 presents the outcome of our feature and requirement identification phase. For the Electric Water Heater and Microwave projects, both LDA-based TF-IDF and BERT demonstrated efficiency, even though different features were identified. The Electric Water Heater had a greater number of functions, possibly due to its complexity, compared to the Microwave. In our case, we consider the top five topics to analyze where the number of words for each topic is set to two.

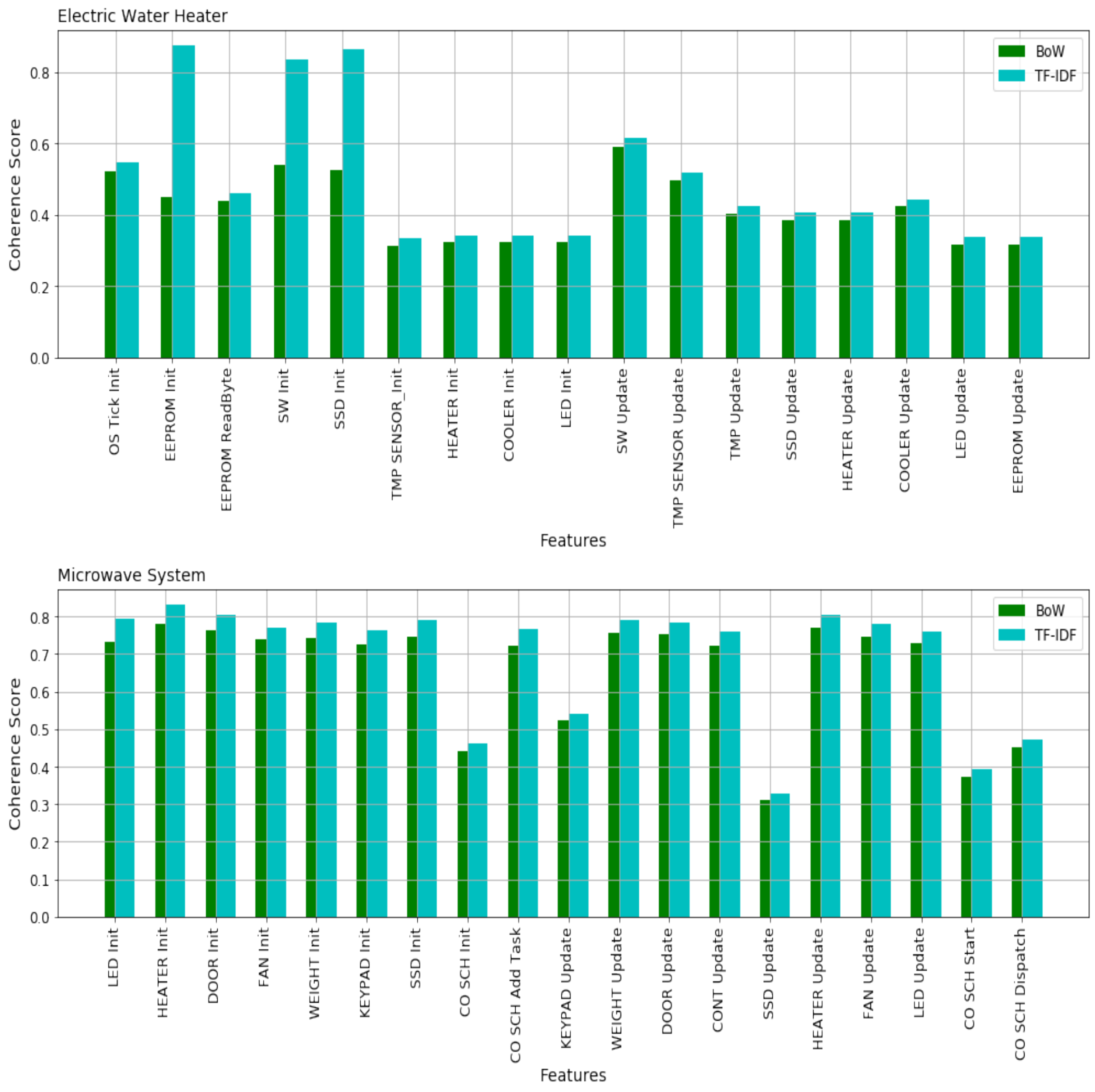


Figure 3.7: Feature coherence evaluation for electric water heater and microwave using LDA+TF-IDF on GitHub projects

Case Study 1: Electric Water Heater

The Electric Water Heater software's analysis shed light on the LDA-based TF-IDF methodology's competence in highlighting essential features. The disparity of coherence between the BoW and TF-IDF methods

Table 3.2: Evaluation of selected features for electric water heater

Features	**Associated Top Functionality**
OS Tick Init	INT Global Init, SET BIT
EEPROM Init	I2C Master Init, SET BIT
EEPROM ReadByte	SET BIT, I2C Read Byte, I2C Write Byte, GET BIT
SW Init	GPIO SetPinDirection, SW Init
SSD Init	GPIO SetPinDirection, SSD Init
TMP SENSOR_Init	ADC Inite, ADC Inite
HEATER Init	GPIO SetPinDirection
COOLER Init	GPIO SetPinDirection
LED Init	GPIO SetPinDirection, LED Off
SW Update	SW Update Period
TMP SENSOR Update	ADC Start, ADC Read Buffer
TMP Update	TMP Required Update, TMP Sensed Update, HEATER Set
SSD Update	SSD Update Mode, SW Period Ended, SSD Off
HEATER Update	LED Set Mode, HEATER Update Mode
COOLER Update	COOLER Update Mode
LED Update	LED On, LED Blink, LED Off
EEPROM Update	I2C Start, EEPROM Write Byte

shown in Figure 3.7 underlines this fact. One of the features, `EEPROM Init`, showcased a coherence score of 0.4481 using BOW, which improved significantly to 0.8750 using TF-IDF. The functionalities linked to this feature, namely `I2C Master Init` and `SET BIT`, indicate functional roles that closely align with the initialization of `EEPROM`, portraying a direct relationship and relevance. Table 3.2 offers a detailed insight into the identified features of the electric water heater, along with their associated top functionalities.

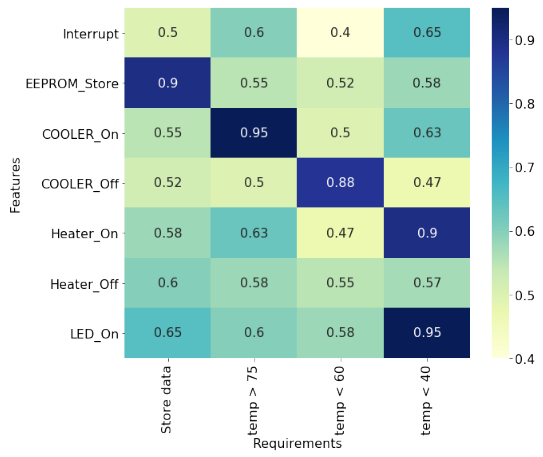


Figure 3.8: Cosine similarities between Features and requirements for electric heater

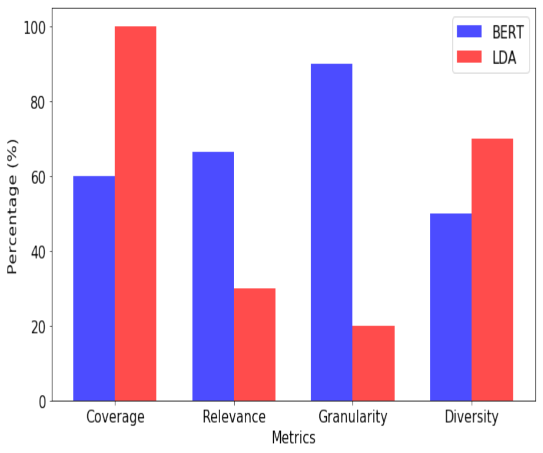


Figure 3.9: Comparison of BERT and LDA based on different metrics

Case Study 2: Microwave

Table 3.1 presents another case study for analyzing Microwave software and its corresponding features and requirements. From Figure 3.7, we find that a majority of the features exhibit higher coherence scores when using the TF-IDF method as compared to BOW. This signifies that TF-IDF provides a more refined and precise measure, particularly in finding important terms within the dataset. For instance, the `LED Init` feature improved from a coherence score of 0.7324 with BOW to 0.7932 with TF-IDF. Similarly, noticeable uplifts in coherence can be observed for features like `HEATER Init`, `DOOR Init`, and `FAN Init`. The bars depict a general trend wherein TF-IDF exhibits superior coherence scores, proving its efficacy over BoW.

BERT-enabled Feature and Requirements Identification

We used BERT, specifically the CodeBERT model designed for code-related tasks, for a detailed analysis. Figure 3.8 shows the similarities between various features and requirements for the Electric Water Heater. A key point to note is that higher similarity values mean a stronger link between the feature and the requirement. With the help of CodeBERT, we were able to understand the code better, which improved BERT's accuracy in identifying features and requirements.

Furthermore, using BERT helped us see connections in the code that might be missed by LDA. This was particularly useful when identifying the differences between functional and non-functional requirements, giving us a clearer view of the software's functions.

BERT vs. LDA

In Figure 3.9, we compare the outcomes of BERT and LDA across several metrics—Coverage, Relevance, Granularity, and Diversity. Each metric offers a unique lens through which the methods' efficiency can be evaluated:

- **Coverage:** LDA excels by capturing the complete set of features or requirements, with BERT trailing behind.
- **Relevance:** BERT shines with a higher score, suggesting that it can identify more contextually pertinent features or requirements than LDA.
- **Granularity:** Here, BERT stands out. It indicates the method's capability to determine detailed or nuanced features and requirements, more so than LDA.

- **Diversity:** LDA tends to be more diverse, identifying a wider range of features or requirements compared to BERT.

Through our experiments, we observe that:

1. Both LDA-based TF-IDF and BERT hold merit, with each being suitable for different contexts and needs.
2. TF-IDF generally offers higher coherence scores, suggesting its effectiveness in feature extraction.
3. BERT tends to be more granular and relevant, whereas LDA shines in terms of coverage and diversity.

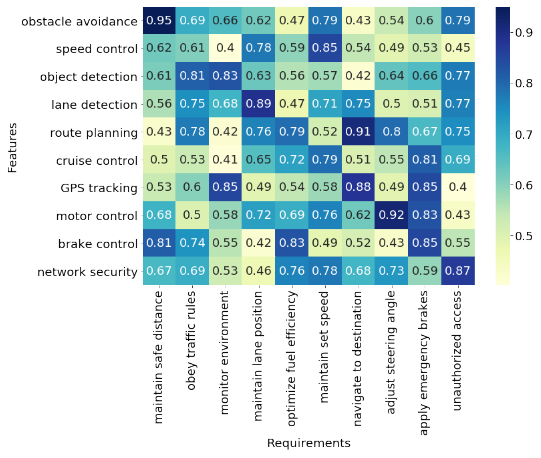


Figure 3.10: Features and requirements mapping for AVS

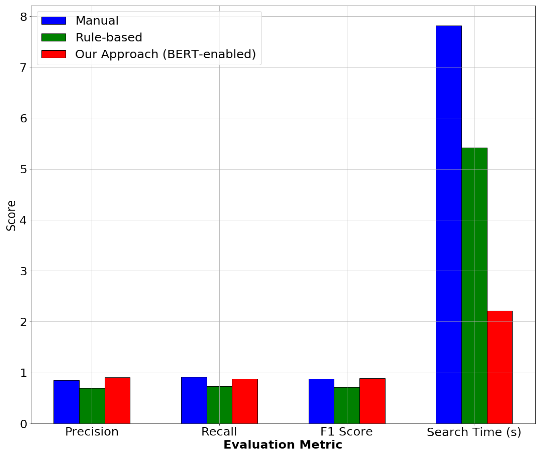


Figure 3.11: Comparison of BERT with Rule-based approach

3.4.2 Comparative Analysis with Other Approaches

We applied our proposed BERT-enabled Topic Modeling LDA Approach to identify different features for the Autonomous Vehicle System (AVS) [114]. Our goal was to compare our BERT-Enabled approach with traditional techniques: the manual and the rule-based methods. In the manual method, the codebase is examined step by step, whereas the rule-based method uses set rules to automatically find features and requirements.

For the AVS, our method pinpointed approximately 65 features along with 180 functional and 30 non-functional requirements. The heatmap of our experiment, shown in Figure 3.10, displays some of the features we identified and their requirements through BERT and topic modeling. We noticed that certain features, such

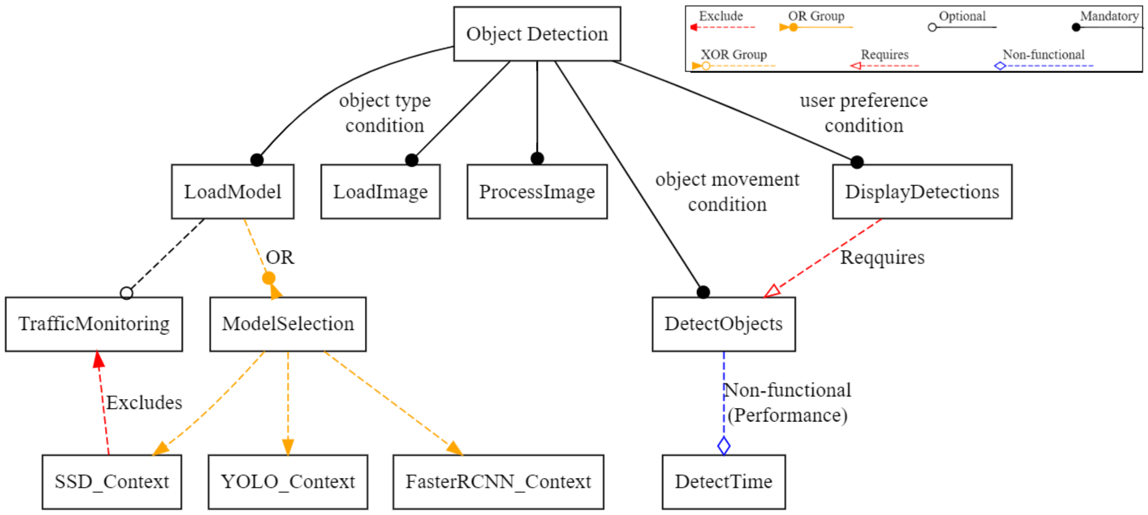


Figure 3.12: Feature model for object detection

as `lane detection`, `object detection`, and `speed control`, were frequently mentioned in the source code. On the other hand, `obstacle avoidance` had various functional requirements. Additionally, the `network security` feature was linked with the non-functional requirement of `unauthorized access`.

Our evaluation employed precision, recall, and F1-score metrics to assess the effectiveness of our approach. Specifically, we achieved a precision of 0.92, a recall of 0.88, and an F1-score of 0.89. These results highlight the balanced performance of our method in terms of both precision (correctness of the identified features) and recall (completeness in identifying relevant features), leading to a high F1-score that effectively balances these two aspects. This score surpassed the ones from the other two methods, illustrated in Figure 3.11. Our method also reduced the time taken to search for features by 59% and 71.7% compared to the rule-based and manual methods, respectively. These improvements are attributed to BERT's advanced language understanding capabilities, which enable more precise and quicker feature extraction from embedded software code than manual or rule-based approaches.

The feature model (Figure 3.12) shows the Object Detection System's architecture. At its core is `Object Detection`, branching into essential features like `LoadModel`, `LoadImage`, `DetectObjects`, and `DisplayDetections`. The `DisplayDetections` feature is dependent on `DetectObjects`, ensuring objects are detected before being displayed.

The `LoadModel` feature offers optional components, such as `TrafficMonitoring`, and provides multiple model selection options through an 'OR' relation among `SSD_Context`, `YOLO_Context`, and `FasterRCNN_Context`. An exclusion link between `SSD_Context` and `TrafficMonitoring` indicates they cannot be active simultaneously. The system also emphasizes performance metrics, marking detection time under `DetectObjects` as a non-functional requirement.

This comprehensive feature model for the Object Detection System provides valuable insights into the software's structure and dependencies. By understanding these relations and conditions, developers can ensure a cohesive design and avoid potential conflicts, especially when making updates or expansions to the system.

3.5 Summary

This chapter delved into combining advanced Natural Language Processing (NLP) techniques with conventional software engineering practices. The primary aim was to identify software features within C code using a combination of BERT and LDA. Software reuse is of paramount importance in developing Cyber-Physical Systems (CPS), given its advantages like cost efficiency, expedited deployment, and risk mitigation. However, finding reusable features from legacy repositories, especially GitHub, poses challenges. These arise due to the intricacies of domain-specific knowledge and often the paucity of comprehensive documentation. To navigate these challenges, we introduced an automated approach that binds static code analysis with deep learning (leveraging the BERT model) and topic modeling techniques (using TF-IDF in tandem with LDA). This methodology is threefold in nature:

1. An extraction algorithm grounded in static code analysis to identify potential features, requirements, and their interrelationships.
2. A semantic discernment module, aided by the BERT model, to grasp the code's semantic nuances and associated documentation.
3. A topic modeling segment employs TF-IDF and LDA to distill the structural core of the codebase.

Our approach showcased respectable performance and resilience when juxtaposed against manual and rule-centric methods on real-world CPS repositories. This underscores its efficacy and hints at its promise in fostering software reuse and refinement in the realm of CPS.

Chapter 4

Next-Generation Embedded Applications Design

4.1 Introduction

The continuous progression of embedded systems technology presents both opportunities and challenges. Embedded software, integral to devices ranging from smartphones to advanced vehicular systems, stands at the center of these developments. Meeting the varied demands of next-generation embedded applications requires the strategic design of sophisticated features into their software base.

A key paradigm facilitating this design process is fog computing. Traditional computing models often relied heavily on central servers, which, while powerful, introduce significant delays due to data transmission and processing times. This can be a barrier for applications where time is of the essence, like vehicular systems. Fog computing offers a solution by decentralizing data processing, bringing it closer to the data sources. This means processing data at the edge of the network or even on the devices themselves, thus reducing latency. However, the successful implementation of fog computing necessitates efficient resource management. Without strategically developed algorithms that adjust resources in alignment with real-time conditions, the system may encounter augmented communication delays or inefficient resource utilization.

Alongside fog computing, the design of Deep Neural Networks (DNNs) within fog and edge devices has gained traction. The capability of DNNs to efficiently process vast datasets renders them a good choice for modern smart devices. Nevertheless, the deployment of extensive DNN models on edge devices, which are often constrained by computational capacities, is challenging. This challenge has led to the development of methods that partition DNNs, distributing the computational demands across multiple devices. Yet, achieving optimal partitioning remains intricate due to various factors, including DNN layer interdependencies and the specific architectural constraints of edge devices.

Furthermore, as edge devices become increasingly autonomous, ensuring the safety of online model training and inference is paramount. The real-time adaptation requirement introduces complexities associated with online model training. Methodologies such as triple-modular redundancy (TMR) have been proposed to navigate these complexities while ensuring system reliability and integrity.

This chapter will explore the aforementioned topics in-depth, emphasizing the relationship between fog computing and edge networks. Edge computing operates as a subset of the broader fog computing paradigm. While fog computing spans from cloud servers to edge devices, edge computing specifically emphasizes the devices themselves. This relationship is key to understanding the importance of next-generation embedded application design in modern software ecosystems. The subsequent sections will offer detailed insights, supported by empirical case studies and data, highlighting the prevailing challenges, proposed solutions, and prospective directions for embedded software in a rapidly evolving technological environment.

- **Enabling Fog Computing for Next-Gen Embedded Applications:** This segment focuses on the deployment strategies for fog-enabled applications, with an in-depth examination of OTA software updates. The system model will illustrate the deployment infrastructure, followed by a methodology that describes the deployment process and subsequent performance evaluation.
- **ML-based Fog Assisted Embedded Application Design:** This section addresses the importance of parallel computing models, aiming to optimize training and inference processes. The system model will provide a schematic representation of the parallel architecture, supplemented by a methodology that explains the mechanisms of achieving enhanced parallelism.
- **Safe Online ML Model Training:** Emphasizing the importance of safety in online model training, this section introduces the Triple-Modular Redundancy (TMR) technique. A robust system model will present the safety layers and protocols, with the methodology elaborating on the practical implementation of TMR.

To understand how the next-generation applications can leverage the advanced computing paradigms, we show an architectural overview to integrate them. The architecture, as illustrated in Figure 4.1, is constructed on multiple layers and components:

Cloud Computing: Centralized cloud service providers (e.g., AWS, Microsoft Azure) that host the primary servers, making them the epicenter for resource intensive computations. It can also house aggregated data or model updates relayed from the edge or fog layers.

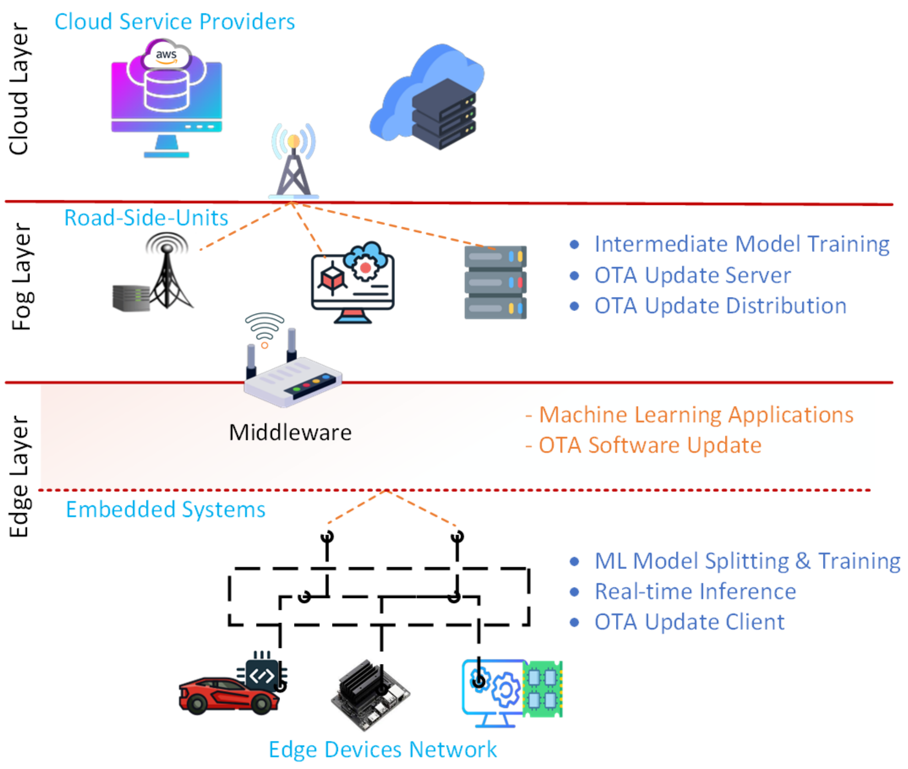


Figure 4.1: Next-generation embedded applications computing architecture

Fog Computing Nodes: These nodes, strategically placed between the cloud and edge devices, serve two primary purposes. These nodes undertake a part of the model training using the technique of model splitting. They also store intermediate ML model states and are responsible for distributing OTA updates to the edge devices. The fog layer helps in reducing the latency for real-time tasks and lessens the computational burden on edge devices.

Middleware: This layer ensures seamless communication between the fog nodes and the embedded software, enabling applications such as real-time insights from machine learning and the facilitation of OTA updates. Furthermore, this middleware aids in the customization of software solutions. Depending on the requirements, specific functionality can be incorporated or excluded to optimize the software for its intended purpose, especially when these applications are designed to run on fog or edge platforms.

Edge and Embedded Devices: These are the actual devices at the far end, interfacing directly with the environment or user. They can either process data locally or send it up the chain to the fog nodes or even the cloud, depending on the computational and latency requirements. The integrated next-generation embedded applications in these devices ensures that data processing and decision-making can happen in real-time.

Summary: The training of machine learning and over-the-air software updates, orchestrated by fog computing, indicates a new generation of embedded software. This integration ensures that the software remains adaptive, intelligent, and customizable, catering to the dynamic needs of modern applications.

4.2 Enabling Fog Computing Architecture for Next-generation Embedded Applications

To select a computing platform closer to the embedded systems and offload tasks to a decentralized infrastructure, we need to design an framework for explicit and well-formalized operations. The next-generation ML-based embedded applications have different characteristics to enable safety, security, mobility, resource optimization, and reliable performance for computing tasks. Considering this, we propose a fog-assisted architecture depicted in Figure 4.2. The architecture presents an over-the-air vehicular software update solution as a case study for next-generation embedded application design. The OTA software update process shows how the fog-assisted architecture manages fog computing resources and helps autonomous vehicles to receive software updates with faster computation compared to the cloud. To implement an OTA update process for next-generation software, we divide our approach into the following steps:

a) Calculate required computational resource: We compute the demanded resource by formulating a traffic cluster in a region at different time intervals.

b) Fog computing resource distribution and optimization: We present a dynamic resource allocation approach to assist in assigning resources when needed.

c) Predicting the communication delay: A transfer learning approach is proposed to estimate the communication delay for making an early decision in selecting the suitable fog node for computation.

d) Calculating the OTA update time: An algorithm is presented to calculate an OTA update time considering mobility, handover, and communication delay.

4.2.1 Fog Computing for Time-Sensitive Embedded Applications

Cloud computing has established itself as a premier choice for the data processing and storage of contemporary applications. However, the considerable propagation delay between the cloud data center and data generation sources might not always be the best solution for time-sensitive applications. On the contrary, fog

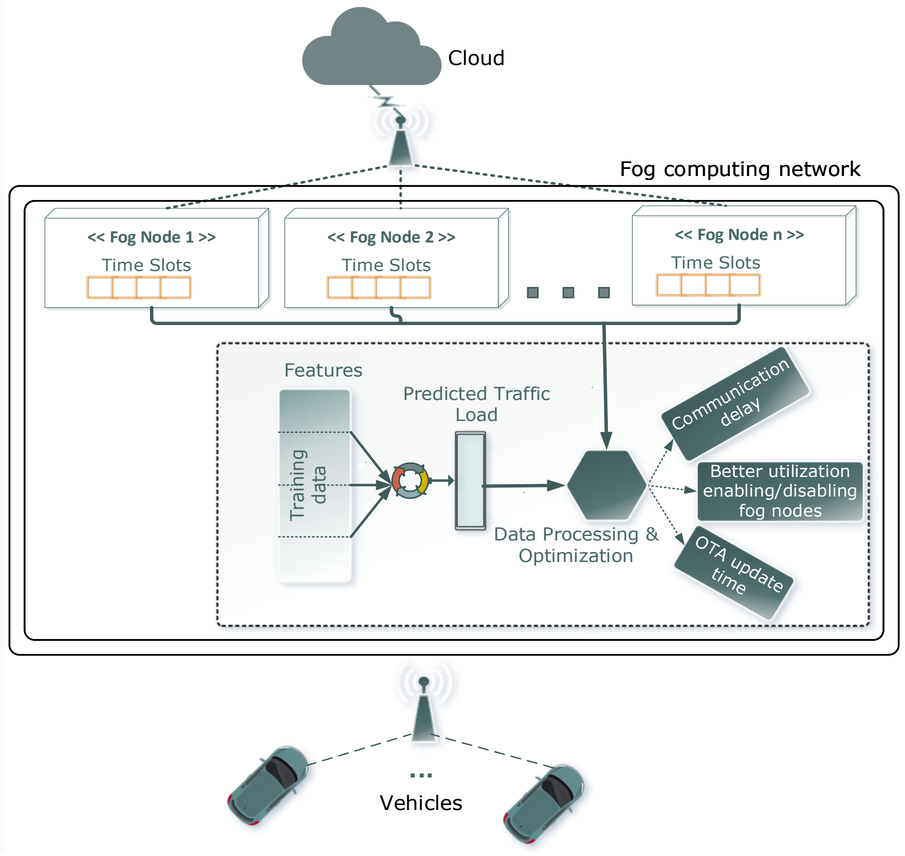


Figure 4.2: Fog computing enabled system architecture

computing has surfaced as a promising alternative for immediate data processing by minimizing communication delays between nodes and data sources. Even with these benefits, establishing an effective infrastructure for fog node management, especially focusing on fog resource allocation, remains an active area of research. Some efforts have been made in this direction, such as the one by Hou [115]. However, many existing works do not focus on resource allocation influenced by traffic load, leading to sub-optimal utilization of fog nodes and, consequently, elevated deployment and running costs.

An application that stands to benefit from an optimized fog computing platform is OTA software updates for intelligent vehicles [23]. Modern vehicles necessitate periodic software updates, which encompass a variety of modifications, from firmware revisions to the introduction of novel features. As indicated by recent survey results [116], the adoption and reliance on OTA software updates are growing, with industry giants like Tesla at the forefront. However, the conventional approach of downloading these updates from vehicle manufacturer cloud data centers presents challenges. The evident propagation delay and the accompanying security vulnerabilities are significant deterrents.

This backdrop brings our work into focus. Our efforts highlight and address the inherent challenges associated with fog computing, especially in the context of OTA updates. These challenges span communication delay prediction, the dynamics of vehicle mobility, and the intricacies of handover delay calculations. In real-world scenarios, these delays can be variable, influenced by a plethora of network and system parameters. Any inaccuracies in accounting for these delays can result in extended OTA update times, potentially leading to compromised system performance. One way to navigate these challenges is to adopt predictive strategies that can closely approximate communication delay, as evidenced by studies such as the one by Kim [117].

Furthermore, the mobility of vehicles introduces another layer of complexity. As vehicles traverse, they engage with a multitude of fog nodes, each potentially offering OTA updates. Transitioning between these nodes, especially transferring the context of software from one node to another, known as a handover, can be resource-intensive. Efficient handovers are crucial to curtailing OTA update times. While newer network architectures like 4G and 5G promise reduced handover latencies, they might not be the silver bullet for OTA updates. Challenges such as the influence of network operators, sporadic packet losses, overheads due to signaling, and prohibitive costs can be limiting factors [118], [119].

Our work presents a comprehensive OTA software update approach that accounts for these challenges and provides solutions centered on efficient fog node distribution and precise OTA update time calculations. By leveraging a combination of machine learning, clustering techniques, and transfer learning, our approach optimizes resource allocation, forecasts traffic loads, and accurately predicts communication delays. This ensures that vehicles, irrespective of whether they are stationary or mobile, can receive their OTA updates swiftly and securely.

In light of recent advancements, such as Tesla's use of the Starlink Satellite Internet for simultaneous OTA update delivery [120], the significance of our work becomes even more pronounced. By enabling continuous OTA updates in dynamic environments and accounting for variables like traffic load, network parameters, and communication delay, our work provides a novel perspective on the realm of connected vehicles and embedded software. The overarching aim is to offer robust solutions that can enhance the efficiency and security of OTA software updates.

Table 4.1: Symbol description

Symbols	Definitions
C_i	Cluster number i; $1 \leq i \leq m$; m ∈N
n	Number of fog nodes
F_k	k^{th} fog node where $1 \leq k \leq n$ and n ∈N
U_k^f	Maximum resource utilization capacity of fog F_k
C_i^{size}	Size of Cluster C_i
C_i^f	Capacity of fog node F_i (requests/second)
$N(C_i)$	Number of fog nodes in Cluster C_i
$N'(C_i)$	Number of active fog nodes in Cluster C_i
$V(C_i)$	Number of vehicle requests in Cluster C_i
p_j	Computational power required by vehicle V_j
$A(C_i)$	Number of active fog nodes required to handle $V(C_i)$
S_j^{size}	Software update size for vehicle V_j, j ∈N
Pkt_j^{arrive}	j^{th} packet arrival time to destination, j ∈N
DR_j	Network throughput for software size S_j^{size}
Pkt_j^{send}	j^{th} packet sending time, j ∈N
T_j	Total time taken to update S_j^{size} for vehicle V_j
cd_{ik}	Communication distance between i^{th} vehicle and k^{th} fog node
T_i^p	Propagation/communication delay between i^{th} vehicle and k^{th} fog node
T_i^t	Processing or transfer time
T_i^{lt}	Network Latency
Y_i	Actual target value
Δ_i	Handover delay
Y_i'	Predicted delay value
W_i	Worst-case delay
A_i^p	Accuracy in predicted model
A_i^p	Expected accuracy set by engineer
$\mathbb{Z}+$	Positive Integer numbers

4.2.2 System Model and Key Assumptions

The fundamental assumption of our research revolves around a fog computing-based network designed for specific regions and varying time intervals. Acknowledging that traffic load demonstrates fluctuations over time across different areas, our model envisions regions with multiple clusters of vehicles. Each of these clusters consists of a variable number of fog nodes. An important assumption here is the visualization of each area as a circular range. The maximum distance from its central position to the farthest traffic location is calculated to determine this range.

We start our experiments leveraging a real-traffic dataset. However, as time progresses and demands shift, this dataset is updated using fresh traffic data collections. A centralized clustering approach serves as the foundation for calculating the OTA update time for location-aware vehicles. In this paradigm, vehicles connect to the nearest fog node, marked by minimal distance and robust signal strength. The central fog node shoulders the responsibility of all computational processing. It's worth noting that while the fog nodes are stationary in their distribution, their activation status is dynamic, contingent upon the update requirements. The centralized methodology aids in pinpointing the cluster size, which is influenced by the number of vehicles in specific locations.

Our architectural model focuses on two important aspects:

1. Efficient distribution of fog nodes across different areas with varied traffic loads.
2. Precise estimation of the OTA update time for vehicles transitioning between locations.

Key Symbols and Resource Allocation Modelling

A comprehensive understanding of our system architecture necessitates familiarity with the various symbols and their corresponding definitions, as showcased in Table 4.1. Exploring further the intricacies of our model, we recognize that each traffic zone is represented by C_i, for every i in m. m is an element of the set of positive integers, $\mathbb{N}^+$. The size of each cluster is denoted by C^{size}, with C_i^{size} representing the traffic volume of cluster C_i.

The centroid of each cluster is identified by $c\_centroid_i$, and the radius of each cluster is c_radius_i. This is measured as the maximum distance between a vehicle position and its cluster centroid position. The range c_range_i is defined as the circular area of a cluster from its centroid position.

In our model, we assume that the number of fog nodes is n. Formally, the set of fog nodes is denoted by F, where F_i becomes the i^{th} fog node. Note that here, $1 \leq i \leq n$ and $n \in \mathbb{N}^+$. Moreover, the set of capacities of all available fog nodes is defined as C^f, where $C^f = \{C_1^f, C_2^f, \ldots, C_n^f\}$. The distribution of these fog nodes needs to be performed across clusters, with the requirement that there is at least one fog node in each cluster.

A uniform distribution is followed while placing fog nodes, so as to cover the total range of each cluster.

Based on the fog node utilization capacity and traffic load, clusters may contain many fog nodes. Let the number of fog nodes in a cluster C_i be given by $N(C_i)$. Then, the following needs to be satisfied: $N(C) = N(C_i) + N(C_{i+1}) + \ldots + N(C_n)$, where $N(C_i) \neq 0$. The number of requests/second that a fog node can handle defines its computational capacity. The assumption here is that vehicles can generate many requests/second. A request is represented as a time requirement. The quantity U_k^f defines the resource utilization of the k^{th} fog node. Next, we calculate the maximum utilization of the k^{th} fog node, at time = t, as follows:

$$U_k^f(t) = \frac{\sum_{j=1}^{V(C_i)} p_j(t)}{C_k^f} \tag{4.1}$$

Now, the utilization for all nodes can be calculated as follows:

$$U(t) = \sum_{k=1}^{n} U_k^f(t) \tag{4.2}$$

Assume that cluster C_i contains $N(C_i)$ nodes. Let $A(C_i)$ denote the number of active nodes that are needed for taking care of requests of smart vehicle $V(C_i)$. Here, $1 \leq A(C_i) \leq N(C_i) \in \mathbb{N}$. Note that the rest of the fog nodes shall remain inactive, till there is additional traffic in C_i. The sum of the computational capacities of individual fog nodes that reside in a cluster C_i defines the computational capacity of the cluster.

For the sake of simplicity, we assume that vehicles generate singular requests, and that all requests need an equal amount of time. Furthermore, each request is for an OTA update. Let the computation requirement of a vehicle V_j be given by P_j, $\forall j \in V(C_i)$. In addition, let C_k^f denote the capacity of the k^{th} fog node, $\forall k \in A(C_i)$. In this case, the following needs to hold:

$$\sum_{j=1}^{V(C_i)} p_j \leq \sum_{k=1}^{A(C_i)} C_k^f \text{ such that } i \in m \tag{4.3}$$

Note that here, $A(C_i) \leq N(C_i)$. Equation 4.3 advises us to fit in $V(C_i)$ vehicles under $A(C_i)$ active fog nodes. In addition, this equation provides an interesting insight: $\sum_{k=1}^{N(C_i)} U_k^f(t) - \sum_{k=1}^{A(C_i)} U_k^f(t)$ of the resource utilization and $N(C_i) - A(C_i)$ of the fog nodes shall be unused in C_i. Therefore, these fog nodes must be kept inactive until we see more traffic in C_i.

OTA Update Time Calculation

The essence of our analysis revolves around the OTA update time calculation. As illustrated in Figure 4.2, vehicles connect to fog nodes via a wireless connection, while the cloud maintains a direct connection with the fog nodes. Our model envisages scenarios wherein the k^{th} fog node, F_k, broadcasts an OTA software update to various vehicles in V.

Our primary objective is to compute the OTA update time, denoted by T_i. This time encapsulates various components: T_i^t (the update transfer time), and T_i^p (the update propagation time). Notably, T_i^t is calculated as the ratio of update size S_i^{size} to the network bandwidth bw_i. T_i^p denotes the time taken for the first bit from F_k to reach V_i. Our approach leverages transfer learning to estimate T_i^p.

The total update time for V_i is - $T_i = T_i^t + T_i^p + \Delta_i$. Note that Δ_i denotes the delay due to handover. In addition, we calculate network latency (T_i^{lt}) and throughput (DR_i). These are calculated from the data packet sending time (Pkt_i^{send}) and the arrival time (Pkt_i^{arrive}) for updating any software size S_i^{size}. The latency shown in Equation 4.4 specifies the total time taken, including processing and propagation for all the packets to transfer from fog nodes to a vehicle, whereas the average throughput refers to the amount of successfully transferred data within a certain period of time in the network.

$$\textit{Latency, } T_i^{lt} = \sum_{j=1}^{x} Pkt_{arrive}^j - Pkt_{send}^j \tag{4.4}$$

$$\textit{Avg. Throughput, } DR_i = \frac{\sum_{j=1}^{x} Pkt_{size}^j}{\sum_{j=1}^{x}(Pkt_{arrive}^j - Pkt_{send}^j)} \tag{4.5}$$

Transfer Learning for Communication Latency Prediction

Transfer learning is a technique that capitalizes on the knowledge acquired while addressing one problem, applying it to a different but related challenge [121]. Unlike traditional machine learning models, transfer learning models train for one problem and subsequently apply the learned knowledge to related problems. The salient advantages of transfer learning include reduced training time, minimized data requirements for training, and enhanced accuracy.

In the context of our work, the potency of transfer learning emerges in predicting the communication delay between vehicles and fog nodes. The European coverage data of the WiFi hot-spot signal strength dataset [122] serves as our starting point. To make the NYC dataset [123] conducive for transfer learning, we enriched it with communication delay data gathered from real-time observations. Our methodology employs a deep neural network machine learning model to forecast the communication delay. Figure 4.3 offers a bird's-eye view of the transfer learning model deployed.

4.2.3 Proposed Fog-Assisted OTA Update Approach for Vehicular Networks

Based on the architecture depicted in Figure 4.2, we first create various clusters, according to the different time intervals of diverse regions. The size or load of each cluster is then calculated by determining the

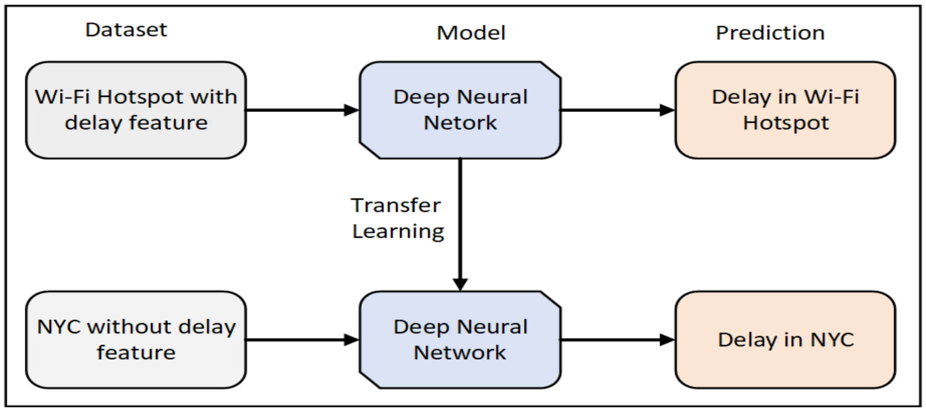


Figure 4.3: Transfer learning model

number of vehicles that exist in the cluster. Each cluster possesses a center point, called its centroid. From this centroid, we define the cluster's range, calculating the maximum distance (as a radius) of a vehicle position.

The steps of the proposed algorithm are as follows: (a) cluster formulation, (b) fog node distribution and optimization, (c) predicting the communication delay and (d) calculating the OTA update time. The OTA update time combines the predicted propagation delay, handover delay and transmission delay for a vehicle that travels from one location to another. Algorithm 4 shows the calculation of the OTA update time in the proposed scheme. We now discuss the details of each step of the proposed algorithm.

Cluster Formation

In order to view different areas in the form of clusters, we divide a region based on the traffic locations, associating different time intervals in a day. For an area, a traffic data set has been used to create clusters of different traffic patterns. Alternatively, we may split the data set based on various intervals of time. For the purpose of clustering, we may consider a host of features: drop off location, drop off date-time, pickup location, pickup date-time.

Next, we apply the k-means clustering scheme on this data set. The main reasons behind the selection of the k-means are that: it is relatively simple to implement, it suits large data sets, and most importantly, it guarantees convergence. Moreover, our goal is to create location-aware traffic clustering for allocating fog resources dynamically. Each fog node would be assigned to a cluster, and all vehicles in a cluster would download OTA updates from this particular fog node. We form the traffic clusters based on the measure of

Algorithm 4: OTA Update Time Calculation

```
1  Input: $ClusterSet$ C[ ], Pickup, Dropoff, Software Size, Speed.
2  Initialize: c_centroids=[ ], c_radius=[ ], c_range=[ ].
3  for $cluster, c \in ClusterSet$ C[ ] do
4      /* zip iterates the locations to find the centroid*/
5      $c\_centroids[c] \leftarrow list(zip(c\_centers[:,0], c\_centers[:,1]))[c]$
6      $c\_radius[c] \leftarrow \max([subtract(i, c\_centroids[c]$ for $i$ in $zip(x,y)])$
7      $c\_range[c] \leftarrow Circle(c\_centroids[c], c\_radius[c])$
8  end
9  /* get the clusters with distributed fog nodes*/
10 $C^{as}$[ ] = Fog_node_distribution ( )
11 /* Calculate the communication delay using transfer learning
12 $T^{p}$[ ] = communication_delay_predict ( )
13 for Vechicle $V_i$ do
14     $distance, d \leftarrow$ Distance(Pickup, Dropoff)
15     $handover\_delay, \Delta_i \leftarrow Delay(d, speed, C^{as}[], c\_range[])$
16     $T_i$ = OTA_update_time ($\Delta_i$, $S_i^{size}$, $F_k$, $bw_i$, $T_i^{req}$, $T_i^{p}$[])
17 end
```

"closeness" or "similarity", where we use the distance measure to identify the closeness. The k-means algorithm is preferred to implement the distance measure function using the Euclidean distance matrix for creating traffic clusters. Although the k-means clustering has some shortcomings, such as the random selection of the initial cluster and isolated vehicle locations, we use this technique due to its robustness over other approaches, like the density-based approach and expectation-maximization. Moreover, we use the elbow curve method shown in Figure 4.4(a) to identify the initial optimal number of clusters. The k-means score is measured through an objective function that computes the intra-cluster distance relative to inner-cluster distance. The simple implementation and the unsupervised learning approach for location-aware traffic clustering make k-means an effective option for integration into our proposed approach.

The time complexity for each iteration of the k-means algorithm is $O(k*n*t_{dist})$, where k is the number of clusters, n is the number of points and t_{dist} is the time to calculate the distance between two points. The number of generated clusters is m. Each cluster has a diverse number of vehicles in a time interval. The next step involves the random selection of m cluster centers. Using the Euclidean distance criterion, we calculate the distance between the cluster center and each data point. Within a Euclidean space, the length of a straight line between two points can be measured by the euclidean distance. Equation 4.6 illustrates the Euclidean distance between point $A(a_1, a_2, ..., a_D)$ and point $B(b_1, b_2, ..., b_D)$.

$$d(A,B) = \|A,B\| = \sqrt{\sum_{n=1}^{D}(a_{in} - b_{jn})^2} \tag{4.6}$$

Here, D represents the dimensional space. Based on the minimum distance from the data point to all cluster centers, each of the data points is placed in a cluster center.

The objective function to assign each data point x_i to the closest cluster (centroid) is as follows:

$$J = \sum_{i=1}^{M}\sum_{k=1}^{K} w_{ik}\left\|x_i - c\_centers[k]\right\|^2 \tag{4.7}$$

Here, if the data point x_i belongs to cluster k, the value of w_{ik} becomes 1, otherwise w_{ik} will be 0. In addition, $c\_centers[k]$ represents the centroid of x_i data point's cluster. This results in a minimization problem that has two parts: firstly, we need to minimize J w.r.t. w_{ik}, and treat $c\_centers[k]$ as fixed. Secondly, we need to minimize J w.r.t. $c\_centers[k]$, and treat w_{ik} as fixed.

Fog Node Distribution and Resource Optimization

Once the cluster formation is done, we need to distribute the fog nodes in various regions. This fog node distribution is the topic of this subsection.

Initially, the clusters are arranged based on C_i^{size}, their cluster size. Note that this represents the number of vehicles. Next, based on their capacity, the fog nodes are arranged in an increasing order. Now, traffic areas are arranged from high traffic areas to low traffic areas. The idea is that high traffic areas are assigned high capacity fog nodes. Next, using equations 4.1 and 4.2, the total utilization is calculated. We assume here that all computation requests are of equal size. The assignment of fog nodes continues in this fashion, till the total computation requirement is fulfilled by x fog nodes. Now, we proceed to the next traffic region and try to meet its OTA update computation demand. This process continues till we cover all traffic areas using the available fog nodes.

Next, we evaluate the load due to traffic, and the computation capacity of each fog node. This will be needed to determine the optimized number of fog nodes needed per cluster. We define the minimum number of active fog nodes for each cluster below.

$$N(C_i) = f(C_i^{size}) = \begin{cases} 0 & : \text{for } C_i^{size} = 0 \\ x & : \frac{C_i^{size}}{\sum_{i=1}^{x} C_i^f} \leq 1 \end{cases}$$

where, $0 < i \leq n$ and $x \leq A(C_i) \leq N(C_i)$.

The assumption here is that the number of fog nodes available is sufficient to process all the traffic load. In other words, there is no traffic area that is assigned a lesser number of fog nodes than required. Once this step is over, we are left with a preliminary number of fog nodes for each traffic area/cluster.

Algorithm 5: Fog_node_distribution ()

```
   Output: Minimum number of fog nodes in each cluster
1  Cluster, $C[] = \{C_1, C_2, \ldots, C_m\}$;
2  Assigned computation, $C_i^{as}[m][] = 0$
3  Required Computation, $C_i^{req}[m]$
4  Number of clusters = $m$;
5  Total number of fog nodes = $n$;
6  struct cluster contains
7      string name;
8      int size;
9  end
10 struct cluster $C_i^{as}[m]$;    /*$0 < C_i^{as} \leq m$*/
11 for i =1 to $m$ do
12     $C_i^{req} \leftarrow \sum_{j=1}^{V(C_i)} p_j$
13 end
14 $F$=sort($F$, $F + n$, greater<int>());
15 for k =1 to $n$ do
16     for i =1 to $m$ do
17         if $C_i^{req} > C_i^{as}$ then
18             $C_i^{as} \leftarrow C_k^f$
19             break;
20         else
21             continue;
22         end
23     end
24 end
25 return $C^{as}$
```

Next, the same data set that was used before is employed for predicting the traffic load. This load is predicted for each cluster for a specific date and accordingly, the cluster size is recalculated to get the number of active fog nodes that are required. Now, if there is an arrival of a new load in a cluster, we would definitely need more fog nodes to process the OTA updates. Say, the initial cluster size is $>$ the traffic load. In this case, we need a smaller number of fog nodes. We note that the traffic load varies according to many factors. Hence, we observe that it would be wasteful to keep all the available fog nodes in an active state. We advise that the fog nodes that are not being used for a particular time interval may be deactivated into a sleep state. Upon an increase in the traffic, those fog nodes may be activated again. We acknowledge that frequent activating and deactivating will introduce an overhead. However, we observe that traffic patterns are generally predictable. Hence, this overhead will not be too significant. If we observe that the traffic load is $>$ the overall capacity of all available fog nodes, some OTA update requests could be offloaded to the cloud. This will have an effect of decreasing the system overload [124]. As a final step, our proposed method predicts the number of fog nodes and the particular fog nodes that need to be activated and deactivated. We discuss the selection of the optimal number of fog nodes in Lemma 4.2.1.

Lemma 4.2.1. Assume a set of fog nodes F, that is distributed among a set of traffic regions or clusters C on the basis of computational capacity. For each cluster C_i, the number of active fog nodes $N'(C_i)$ is optimal with respect to the initially defined number of fog nodes $N(C_i)$. Therefore, $N'(C_i) \leq N(C_i)$.

Proof. Suppose we have a cluster set, where C_i^{req} is the required computational power for a particular cluster size C_i^{size}. According to Algorithm 5, the distribution of fog nodes for each cluster should be followed by Equation 4.8.

$$\frac{C_i^{req}}{C_i^{as}} \leq 1 \tag{4.8}$$

Therefore,

$$\frac{\sum_{j=1}^{V(C_i)} p_j(t)}{\sum_{k=1}^{x} C_k^f} \leq 1 \quad \text{: where x} \leq n.$$

$$\text{Now, the unused computational capacity,} \, UC_i = C_i^{as} - C_i^{req}.$$

In real-time, the cluster size may either increase or decrease.

Case 1: If the new required computational power ($C_i^{req'}$) becomes less than previous C_i^{req}, the new unused computational capacity UC_i' will be:

$$UC_i' = UC_i + (C_i^{as} - C_i^{req'})$$

If $\frac{C_i^{req'}}{\sum_{k=1}^{l} C_k^f} \geq 1$ and $\frac{UC_i'}{\sum_{k=(l-1)}^{x} C_k^f} \geq 1$, then Equation 4.9 is always true for $l(1 \leq l \leq x)$, the optimal number of fog nodes in cluster C_i.

$$\Rightarrow \frac{C_i^{req'}}{C_i^{as'}} \leq 1 \tag{4.9}$$

Here $C_i^{as'}$ is the total computational capacity of all active fog nodes, which meets the requirement of cluster C_i. Therefore, $(x - l)$ is the number of inactive fog nodes and $N'(C_i) = l$ is the optimal number of fog nodes which is always less than or equal to x.

Case 2: If the required computational power of a cluster increases, the additional required computational power AC_i should be less than or equal to the remaining unused capacity UC_i, as we assume that the initial cluster size C_i^{size} is the maximum size for each cluster. However, if it goes beyond the maximum limit, then we propose to offload the excess computation to the cloud.

Therefore, it is proved that the number of active fog nodes $N'(C_i)$ becomes optimal in both cases.

□

Corollary 4.2.2. If $N(C_i)$ and $N'(C_i)$ are positive integer numbers then

$$\forall x; N'(C_i) + x \leq N(C_i) + x; \text{ where } x \in \mathbb{Z}+$$

Proof. While x is a positive integer number, the range of the value always remains between $[N'(C_i), N(C_i) + x]$. According to the mean value theorem, if *f(x)* is a function (*f(x)* $= N(C_i) + x$) on the closed interval $[a, b]$ and differentiable, then the point c in (a,b) follows the rule given below:

$$f'(c) = \frac{f(b)-f(a)}{b-a}$$
$$\Longrightarrow f'(c) = \frac{N(C_i) + N(C_i) + x - N(C_i) - N'(C_i)}{N(C_i) + x - N'(C_i)}$$

The derivative of a constant value c is zero and the value is a positive integer. Thus,

$$\implies 0 \leq \frac{N(C_i) + x - N'(C_i)}{N(C_i) + x - N'(C_i)}$$
$$\implies 0 \leq 1$$

Therefore, it is proved that the statement of Corollary 4.2.2 remains true for any value of $x \in \mathbb{Z}+$. □

Predicting the Communication Delay

To calculate the OTA update time, we use the transfer learning approach to predict the communication delay. A deep neural network has been used to find the delay bound. The worst-case delay is improvised to ensure the safety bound of the communication delay. To do so, we consider two scenarios. In one scenario, the target domain contains a training and testing dataset with known output. Thus, the target value denotes the actual expected value and the worst-case delay is calculated from the engineer defined threshold value. In another scenario, the target domain data set does not have the actual output. Therefore, we propose to set the target value by taking the average of the actual delays from the training data set and calculate the worst-case delay taking a threshold that depends on the user inputs. Algorithm 6 shows the details of selecting delay values. The Mean Absolute Error (MAE) loss function is used to determine how far a target value is from a predicted value.

$$MSE = \frac{\sum_{i=1}^{m} |E_i|}{m} \tag{4.10}$$

In Equation 4.7 above, E_i is $(Y_i - Y_i')$, where Y_i is the actual value, Y_i' is the predicted value, and m represents the total number of training examples.

We have a source domain D_s with corresponding task T_s and a target domain D_t with target task T_t. Our objective is to learn the conditional probability distribution $P(Y_t|X_t)$ in D_t, using the knowledge retrieved from D_s. We compared the predicted communication delay with the worst case delay d_w. We achieved 88% accuracy on the data-set [122].

Proposition 4.2.3. For a given known domain D_s and a target domain D_t, the task is to predict the communication delay for different locations in D_t. The delay value selection shows the relaxation with an upper bound or worst-case delay for which $d_i \leq W_i$.

Algorithm 6: Relaxed_delay_bound_check()

```
    Input  : One training dataset $DF_s$ and one test dataset $DF_t$
    Output: Make decision in selecting delay value
 1  set the threshold, $\epsilon_i$
 2  set the expected accuracy, $A_i^r$
 3  if target domain has known output then
 4      Target value, $Y_i$ = actual data
 5  else
 6      Target value, $Y_i$ = mean of delays from training data
 7  end
 8   Worst case delay, $W_i$ = $Y_i$ + $\epsilon_i\%$
 9  Initialize the deep neural network N;
10  Train Deep neural network using $D_s$;
11  $Y_i' \leftarrow$ Predicted delay value
12  $A_i^p \leftarrow$ Accuracy in predicted model
13  if $(Y_i \leq Y_i' \leq W_i)$ && $(A_i^p > A_i^r)$ then
14      /*carry on with in-network predicted value*/
15      $d_i = Y_i'$
16  else
17      /*select the worst-case delay*/
18      $d_i = W_i$
19  end
```

Proof. Let us assume y' is the predicted value for the given model using $y' = w.x$, where x denotes the vector of data used for training, and w is the weight. To find the distance between the predicted value and the actual (expected) value, the mean absolute error (MSE) loss function is used.

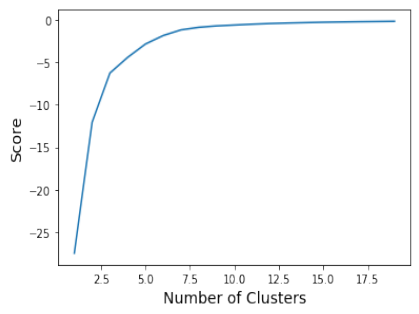
$$difference = |y' - y|$$

Due to the absolute result, the cost function will always return the positive value for which the predicted value will never be a negative number.

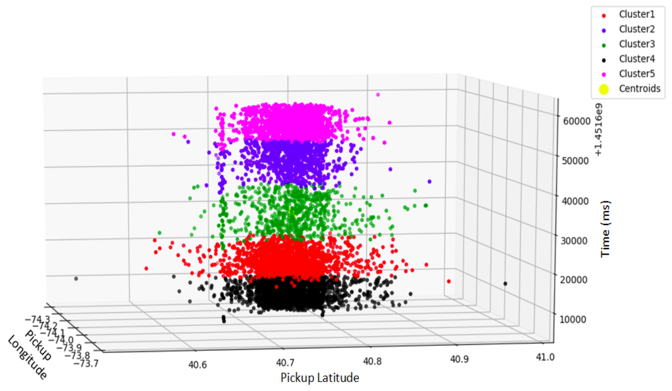
For target dataset,

$$D_t = (x_i, y_i')_{i=1}^m$$

where $y_i' = (y_i, y_1', y_2', \ldots, y_m')^T$, y_i is the real value and $y_1', y_2', \ldots, y_m'$ are the predicted values. Now, according to Algorithm 6, the predicted value $y_i' = Predictor(x_i)$. Here, the target value is set to either y_i, or to the mean of all the delays that are defined in dataset D_s. Therefore, y_i' can be higher or less than y_i. If the predicted delay remains between the targeted delay y_i and the worst-case delay W_i, then y_i' is selected to calculate the OTA update time. At the same time, the accuracy of the prediction needs to be satisfied to ensure the safety bound. However, if the prediction goes out of the user-defined worst-case delay, the proposed algorithm suggests selecting W_i as the delay. Hence, it shows that the algorithm selects the delay value that always maintains the uppermost bound for safe prediction. □



(a) Elbow curve to determine the optimal number of clusters



(b) K-means clustering on NYC taxi dataset

Figure 4.4: Clustering to find the traffic pattern and the number of vehicles

OTA Update Time Calculation

In order to calculate the OTA update time, the proposed approach tracks a vehicle's route between its pickup and drop off location. The vehicle's route helps in identifying the clusters that reside along its path. As a result, the total distance is an indirect indication of how many fog nodes can potentially provide service to the vehicle during a software update. The propagation delay between the vehicle and the fog node is calculated using transfer learning. We use the Wi-Fi Hotspot dataset to predict the delay for any particular location or area. As the propagation varies from one location to another, we propose to transfer the previous knowledge in predicting the delay to a new location. Therefore, we calculate the communication or propagation delay for a vehicle in establishing a connection to a fog network. This helps in selecting fog nodes which incur minimum communication delay while providing updates/connectivity to the vehicles in the cluster.

For the OTA update time calculation, we consider various constraints, and evaluate the proposed fog node allocation scheme. It is the assumption that a software update S_i^{size} in size is needed by a vehicle V_i.

Moreover, the update needs to be delivered within time T_i^{req}. We need to calculate now T_i, which is the total required time. For this, we need to calculate the propagation delays for various traffic locations. We use the transfer machine learning approach for this.

In addition, we also calculate Δ_i, which is the delay due to handover. This quantity is based on the distance covered by the vehicle, and can be obtained from the trip time and speed of the vehicle. Next, within that distance, we calculate the number of fog nodes that are required. We then calculate the handover delay. After that, the total transmission time T_i^t needed for downloading the OTA update of size S_i^{size} is calculated. Next, the total time T_i that it takes to deliver an OTA update of size S_i^{size} to the vehicle is calculated. The OTA update process is described in detail in Algorithm 7. The optimization problem for updating the vehicle software amounts to minimizing the overall OTA update time by ensuring that the propagation delay and handover delay is minimum.

$$Minimize \sum_{i=1}^{n} T_i$$

subject to:

$$\forall i \in n \;\; T_i = T_i^p + T_i^t + \Delta_i.$$

In order to ensure that the overall OTA update time is minimized, the OTA manager will select the fog nodes which offer minimum communication delay. Moreover, the manager will ensure that the number of fog nodes providing the software updates to vehicles is minimized, so as to keep Δ_i low.

4.2.4 Evaluation of Fog-Assisted OTA Software Update Approach

To evaluate the performance of our proposed OTA update algorithm, we conduct several experiments and analyze the results. The broad goals for these experiments are given below:

- Using k-means clustering, we label and envision the various traffic regions, using the rate of flow of traffic.
- Through efficient distribution, find the optimal number of fog resources required for satisfactorily providing OTA updates to vehicles.
- Analyze the OTA update time using Mininet-WiFi, with handover delay and propagation delay as its components. The propagation or communication delays are obtained and integrated from the predicted

Algorithm 7: OTA_update_time ( )

```
   Input : $S_i^{size}, F_k, bw_i, T_i^{req}, \Delta_i$
1  $OTA$ fog manager monitors the fog node utilization
2  while node utilization is less than 1 do
3      $T_i^p = communication\_delay\_predict()$;
4      $T_i^t = S_i^{size}/bw_i$;
5      Calculate the total OTA update time, $T_i = T_i^p + T_i^t + \Delta_i$;
6      if $T_i \leq T_i^{req}$ then
7          Update Manager triggers transition
8          if Update for $S_i^{size}$ failed then
9              $F_k$ restores it
10         else
11             continue with next update
12         end
13     end
14 end
```

transfer learning mechanism that uses WiFi hotspot and 5G datasets.

Optimized Fog Resource Allocation

In order to comprehend the pattern of traffic and to efficiently distribute the fog nodes, we analyzed the NYC taxi drive data set [123] for the month of March (31 days). The following features were extracted from this data set: id, dropoff_datetime, pickup_datetime, dropoff_latitude, dropoff_longitude, pickup_latitude, pickup_longitude, speed, distance, and trip time.

Initially, based on datetime and traffic location, we determine the number of clusters. For each day's data of the dataset, we execute the k-means algorithm to determine the number of clusters and cluster sizes. Consider the elbow curve of Figure 4.4(a). We use this curve to find the optimal number of clusters in the NYC data set. The optimal number of clusters is hence defined as five. We find the maximum size of each cluster post application of the k-means algorithm for each day. Figure 4.4(b) shows the distribution of traffic in each cluster, where each cluster has a different number of vehicles. According to our system model, we assume that each cluster size has the maximum amount of traffic. In addition, we find the centroids of each cluster following Algorithm 4, and calculate the distance of each vehicle from its centroid position. Finding the maximum distance for a vehicle from a centroid in each cluster, we calculate the radius of each cluster. Figure 4.5 shows the range of each cluster and sample points of vehicles in each cluster with the centroids.

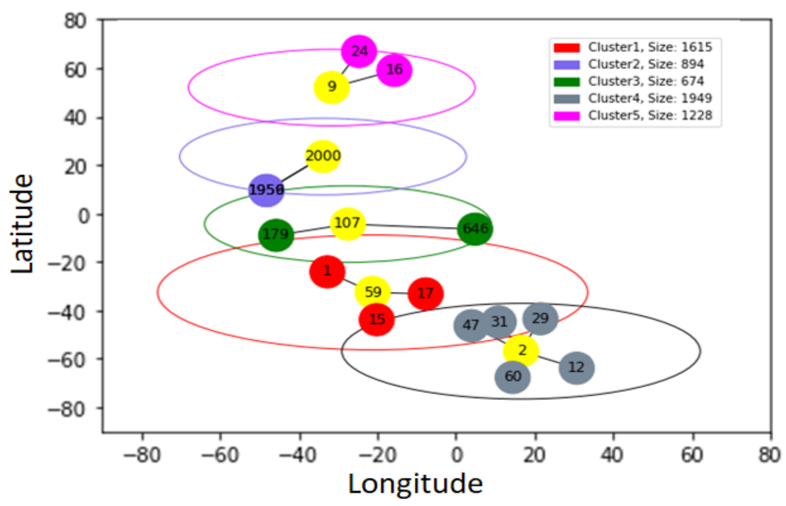
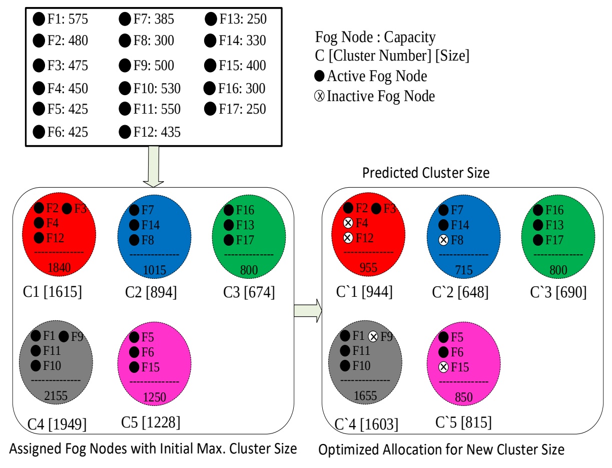
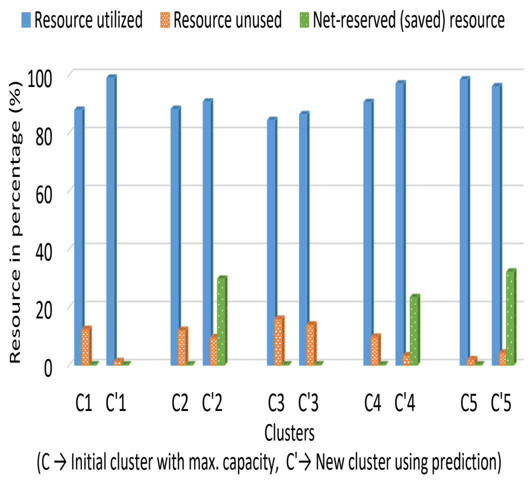


Figure 4.5: Range and size of each cluster



(a) Resource allocation example



(b) Allocated fog nodes utilization in different clusters

Figure 4.6: Fog nodes distribution using Algorithm 2 and resource utilization calculation in different clusters

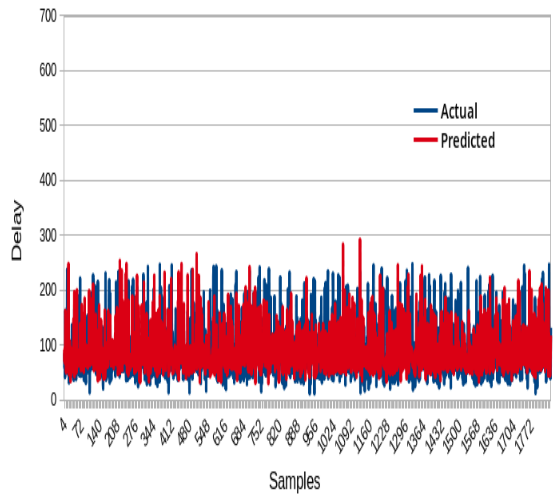
Using Algorithm 5, we calculate the initial cluster size of diverse locations at specific times. Please see Figure 4.6(a) for the results. This experiment was done with the number of fog nodes equal to 17. The capacities of various fog nodes are shown in Figure 4.6(a).

The fog nodes are sorted in a descending order based on their capacities and then distributed among different clusters. This process is carried out till all the cluster requests are satisfied by the fog nodes. We do note that the traffic size in a cluster does not always take the maximum value. Hence, using neural network machine learning, we predict the cluster size. This has the effect of optimizing the fog resources at hand. The assumption here is that the predicted cluster size is not greater than the initial cluster size, leading to a smaller number of fog resources. The number of available fog nodes in a cluster is enough to handle all incoming traffic to that cluster. If the fog nodes do not have enough capacity to accommodate the vehicles in that cluster or if any fog node(s) becomes faulty, the traffic is offloaded to the cloud data center. The proposed framework can handle variable traffic. If the incoming load in any cluster is less than its capacity, then the idle fog nodes will be put on inactive mode. Later on, when the traffic increases, the inactive fog nodes will become active according to the new requirement.

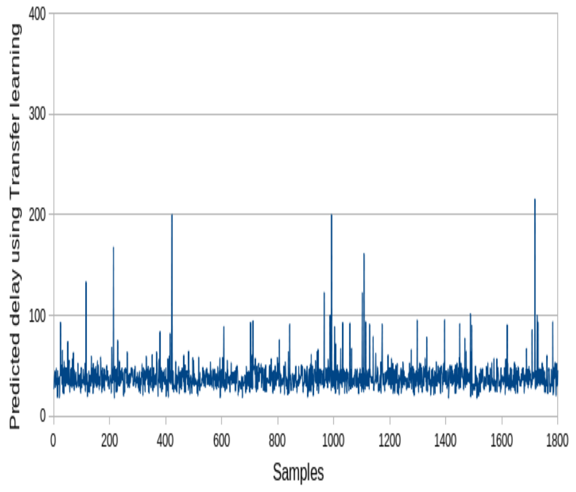
It is our advice that the fog nodes that are not needed to service the requests of the newly sized cluster be kept in an inactivate state. These inactive fog nodes become part of the net reserve of fog resources, as shown in Figure 4.6(a). We note that this optimization of fog nodes maximizes resource utilization and minimizes power consumption. For the next experiment, we define three different indicators in Figure 4.6(b): fog resource utilized, fog resource unused, and fog net reserve. The resource utilized symbolizes the total amount of used resource of all fog nodes in a cluster at a particular time. The unused resource determines the amount of resource that is allocated, but not in use. It is inversely proportional to the resource utilization of fog nodes. On the other hand, the net reserve resource is the actual resource (not allocated) that can be used for other purposes. Therefore, if the resource utilization and unused resource decrease, then the net reserve resources will also increase, and it can be distributed following Algorithm 5. The symbols C1 - C5 correspond to the clusters before application of our proposed algorithms and the symbols C'1 - C'5 refer to the clusters after application of our proposed algorithms. From Figure 4.6(b), we observe that for each cluster, we see an average of 93.69% fog node utilization upon allocation, 6.31% of unused fog resources, and the net reserve (after optimization) of fog resources. We observe that there is a 26.57% increment in the overall net reserve fog resources on average, which reflects good system performance as a result of our proposed approach.

Communication Delay Prediction

In this experiment, we discuss the transfer learning approach to predict the propagation/communication delay, which is used to calculate the final OTA update time for different software sizes, vehicle locations and cellular technologies. We employ a neural network model for this delay prediction.



(a) Delay prediction



(b) Delay in NYC dataset using transfer learning

Figure 4.7: Propagation delay prediction using transfer learning

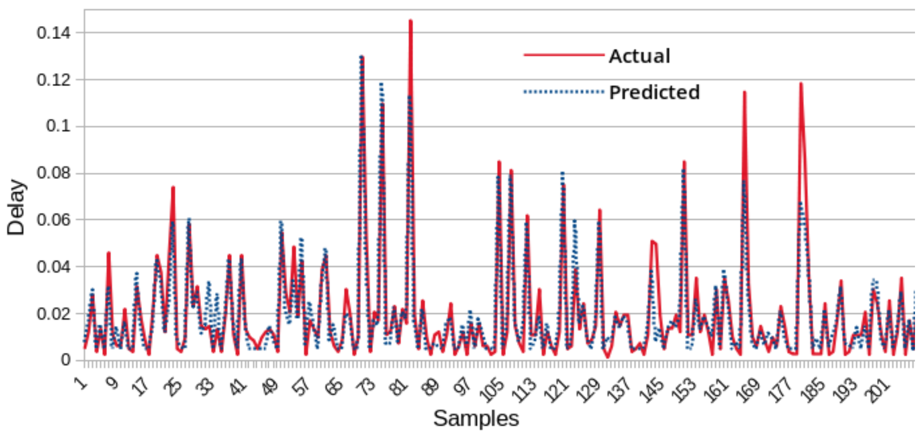


Figure 4.8: Propagation delay prediction using 5G dataset [125]

In order to train the neural network model, we used the European coverage data of the WiFi hotspot signal strength dataset [122] and a 5G dataset [125]. WiFi hotspot signal strength dataset has been collected by a crowd-sourced application called netBravo. This dataset includes a grid shape and a csv file based on the GRID_ETRS86 reference system. We split the dataset into two parts: 75% of the data is used to train the model, while 25% of the data is used to test the model. On the other hand, the 5G dataset is a 5G trace collected from a major Irish mobile operator. This dataset is composed of static and dynamic mobility

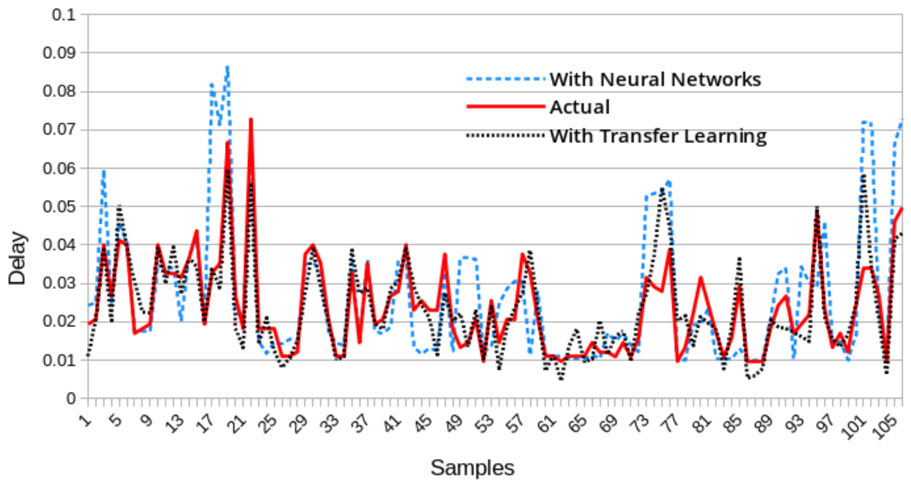


Figure 4.9: Propagation delay prediction with Transfer Learning and Neural Network on 5G dataset

patterns. It has been created on the basis of two applications: video streaming and file download.

In terms of machine learning models, we used Linear Regression, XGBoost, Random Forest Regressor and Deep Neural Networks for training. Among all of these models, the Deep Neural Network demonstrated better prediction results. Specifically, in order to predict the communication delay, we used a deep neural network with three hidden layers, with each hidden layer having 256 nodes. On the output layer, we employed a linear activation function.

The WiFi hotspot signal strength dataset has features such as: x, y, type, upload, download and technology. X and y are the coordinates of the cell grid, type represents whether the connection is WiFi or cellular, technology may be of five types: 2.5 & 5 GHz in case of WiFi and 2G, 3G & 4G in case of cellular. Upload & download are the upload and download speeds respectively, in Kbps.

Figure 4.7(a) shows the performance of our neural network on the European coverage data of the WiFi hotspot signal strength data-set. The red line represents the predicted value, and the blue line represents the actual value. As the figure shows, most of the observations have a delay between 40 ms and 150 ms. In a real time environment, the communication delay mostly follows this interval. In some cases, the delay goes above or below this range. The x-axis represents the dataset samples. Each sample represents a point in the geographical area given in the WiFi hotspot dataset. Figure 4.7(b) shows the predicted communication delay in the vehicular network. The x-axis represents a point in the geographical area given in the NYC dataset. Now, for each coordinate in NYC dataset, we have one predicted value representing the communication delay

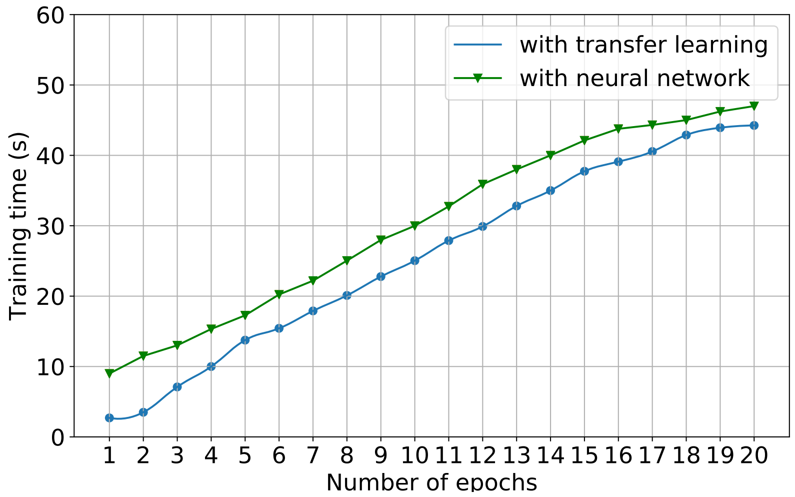


Figure 4.10: Transfer learning performance on training time

between a fog node and the vehicles in its cluster.

Figure 4.8 shows the delay prediction performance using the 5G dataset. Figure 4.9 depicts the performance comparison between the ML models: with and without transfer learning, using the 5G dataset. The Transfer Learning approach has been compared with a neural network model consisting of one input layer, three hidden layers and one output layer. The input layer has 128 nodes, the hidden layers have 64 nodes each and the output layer has 1 node. As shown in Figure 4.9, the transfer learning prediction is much closer to actual values. This is because, for the model with neural network, we had less data to train the model. In the transfer learning model, we used an already trained model, hence there was better training. The accuracy achieved with neural network is 72%, compared to the 81% achieved with transfer learning.

To understand the impact of transfer learning on calculating the OTA update time, we compare it to the regular approach of DNN without transfer learning. Figure 4.10 shows a performance comparison for the training time between the “transfer learning” and the empirical approach. We record the training time for different number of epochs. The dataset is divided into a number of batches (with default batch size 32), and each epoch feeds the required number of batches to pass through the entire training dataset once. The machine learning algorithm updates the internal model parameters in each epoch to improve the model accuracy and training loss. In transfer learning, we add the new training data on a pre-trained model whereas the other approach appends the new data in the earlier training data. In Figure 4.10, it is clearly visible that the transfer learning approach requires lower training time in comparison because of its pre-trained model. The training time shows a consistent difference between these two approaches as transfer learning uses the pre-trained model. However, the difference gradually decreases when the required number of epochs increases for training a larger dataset. Therefore, if the training dataset of the target domain is not too large, the proposed approach with transfer learning will benefit the model in terms of both training time and accuracy.

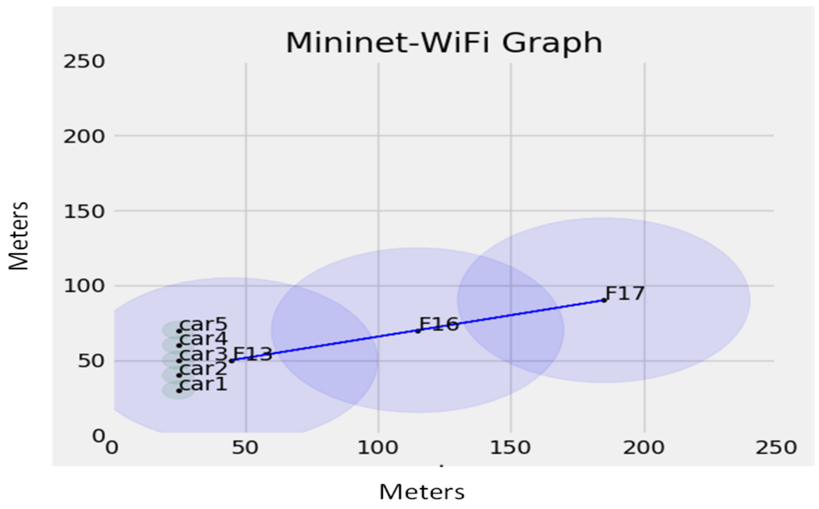


Figure 4.11: Mininet-WiFi network architecture for cluster 3

OTA Update Time Calculation using Mininet-WiFi

This experimental subsection discusses the implementation of the proposed OTA update scheme on the Mininet-WiFi [88], which supports vehicle mobility and wireless communication under the simulated fog architecture. Mininet provides a simple and inexpensive network testbed for developing OpenFlow applications compared to other emulators. The architecture provides the specification of flows following the software-defined network (SDN) and allows us to add a network of virtual hosts, switches, controllers, and links. The flow specification contains the link delay, bandwidth, vehicle speed, network topology, and switch information. The simulation calculates the OTA update time for different software sizes exploring the required handover delay.

This experiment was performed employing five cars/vehicles and three fog nodes in a 250x250 meter area. The Mini net-WiFi network uses the 802.11g standard for transmitting data over a wireless network. The bandwidth of the WiFi-network is 2.4 GHz and the maximum data transfer rate is 50 Mbps. We attempt to find out the OTA update time for software of various sizes within a cluster, which is cluster 3 shown in

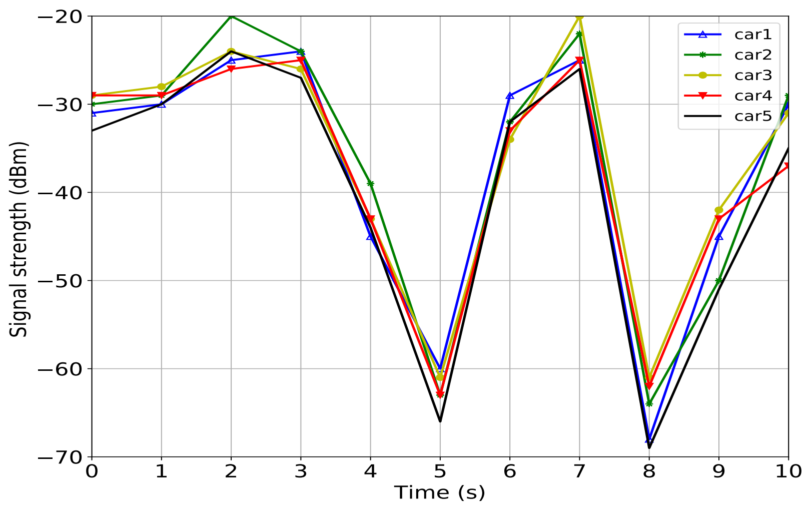


Figure 4.12: Signal strength of each car over the time

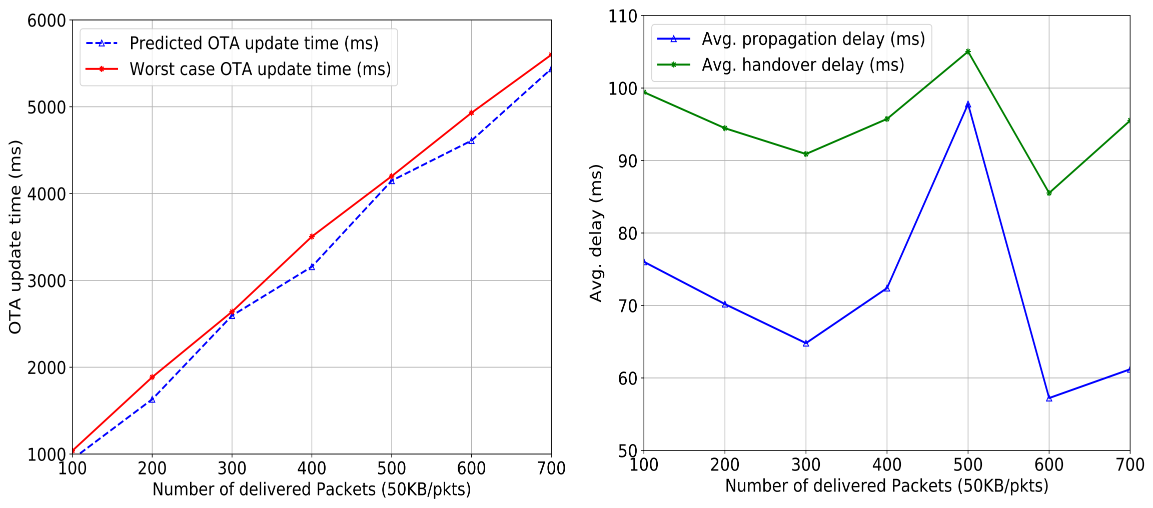
Table 4.2: Mininet-WiFi network parameters

Artifacts/Parameters	Values
Number of cars/vehicles	5
Number of fog nodes	3
Vehicle speed	14m/s or 36km/hr
Bandwidth of links	50Mbps
Predicted propagation delay (ms)	76.030876, 70.1916, 64.80096, 72.4002 and 97.80364
Propagation model	logDistance
Association control	ssf (Strongest-Signal-First)
Range of each fog node	30m
Interface	Wlan0, Wlan1

Figure 4.6(a). The details of the experimental architecture are shown in Figure 4.11.

Our proposed solution evaluates the effectiveness of the predicted communication delay by calculating the handover delay and OTA update time in the presence of multiple traffic requests in the network. Table 4.2 lists all the network parameters used.

For this experiment, we take into account five cars moving at a speed of 14m/s (meters/second) from their pickup locations to their drop-off locations. Each car will be connected to a fog node when it goes under its coverage area. Once the vehicles start moving from one position to another, the signal strength of the fog node connections also varies over time. The ranges of fog nodes overlap where the vehicles select the strongest-signal connection. We find the signal strength after every second for each car, when it changes its



(a) OTA update time for different data size (b) Handover and propagation delay

Figure 4.13: OTA update time and handover delay

position towards the drop-off location. Figure 4.12 shows the variations of signal strength for 10 seconds. During times equal to 5 and 8 seconds, the signal strength goes to its lowest, and each car performs a handover to connect with the strongest fog node signal. Thus, we see a gradual increase in fog node signal strength. This indicates a successful handover in the overlapping region.

Simultaneously, the mobility starting and ending points are added for each car and the speeds of the cars are set at 10m/s. When the cars start moving while following their route, they get connected in the middle of the route with different fog nodes, such as fog node 13, 16, and 17. At the beginning, all the cars remain under the coverage of fog node 13 through interface Wlan0, while the other interfaces remain off. The bandwidth of the links is set to 50 Mbps.

During the time the car is mobile, we calculate the total OTA update time by varying the number of transmitted data packets. We note that owing to variations in the distance, the data transmission rates may fluctuate. We assume that an update consisting of a number of data packets starting from 100 to 500, is pushed from fog nodes to the cars, where each data packet is equivalent to 50Kb. We set the predicted propagation delays in the transmission flow, which are 76.030876, 70.1916, 64.80096, 72.4002, and 97.80364 ms, respectively. Based on the OTA update Algorithm 7, we calculate the data transmission time measuring the time taken to complete the update transfer. As the last step, the propagation time and transfer time are subtracted in order to obtain the handover delay. The handover delay (Δ_i) includes the following delays:

$$\Delta_i = D_{rang} + D_{req} + D_{res} + D_{ex} \tag{4.11}$$

Here,

- D_{rang} = Time required for initial ranging process. This finds out the fog nodes with a maximum coverage area.
- D_{req} = Time required for requesting to connect to a new node.
- D_{res} = Time required to register with target fog node.
- D_{ex} = Time required for message exchange.

In order to make a comparison with the worst-case scenario of propagation delay, we calculate the OTA update time maintaining the same procedure as before. In this case, the worst-case time is selected from the maximum of all the predicted propagation delays.

The results of this experiment are shown in Figure 4.13. Looking at Figure 4.13(a), we see that the OTA update time is 940.31 ms for 100 data packets, and 5437.18 ms for 700 data packets. With the worst-case propagation delay, the OTA update time increases to 1037.34 ms and 5599.45 ms for 100 and 700 packets, respectively. We observe that the average overall OTA update time is reduced by 5.34%.

Figure 4.13(b) shows the comparison of the average predicted propagation delay and the average handover delay for all five cars. When the number of delivered packets varies, the average handover delay changes with respect to propagation delay. We observe from the figure that the handover delay fluctuates, leading to a high value when the propagation delay is high, and low value when the propagation delay is low. The intuition is that the correct prediction of the propagation delay can help to determine the handover delay and overall OTA update time. Therefore, the proposed scheme using transfer learning offers a better solution in calculating the OTA update time for making any decision at an earlier time. It helps to decide whether the OTA update process should proceed or not. The transfer learning approach benefits in training the model to find the expected communication delay for which the OTA update time calculation becomes faster.

We conduct another experiment to see the effect of varying the number of vehicles in the proposed architecture. The number of vehicles was varied from 10 to 100, and the corresponding OTA update time, the sum of propagation delay and handover delay were recorded. The result of this experiment is shown in Table 4.3. The experiment sends 100 data packets for each run where each data packet comprises a 50 Kilobytes

Table 4.3: Effect on OTA update time for increasing number of vehicles

# Vehicles	**# Fog Nodes**	T_i **(ms)**	**Delay (ms)** $(T_i^p + \Delta_i)$
10	3	1060.82	174.5
20	3	1081.43	184.2
30	3	1064.39	163.1
40	3	1095.34	175.3
50	3	1130.10	205.1
60	3	1125.21	169.4
70	3	1136.26	167.6
80	3	1159.77	186.4
100	3	1168.43	175.4
10	4	1067.34	176.79
20	4	1074.67	174.69
30	4	1092.3	190.1
40	4	1106.5	198.57
50	4	1100.28	177.5
60	4	1117.23	186.34
70	4	1128.52	203
80	4	1136.41	204.88
90	4	1149.38	214.13
100	4	1150.54	206.7

of zero inside of it. We observe that the average OTA update time increases when the number of vehicles increases. When the number of fog nodes increases from three to four, the propagation delay decreases, but the handover delay increases lightly. Therefore, the OTA update either remains close to the previous result or starts rising when the number of cars increases. The reason behind the increased OTA update time is an increase in the overall resource utilization and communication delay. With a 95% confidence level in OTA update times listed in Table 4.3, the upper limit of the confidence interval (CI) is 1130.75ms, and the lower limit is 1099.00ms. In the case of delay calculation, the standard mean of delay is 184.64ms, and the error is almost 3.5% where the estimated upper CI is 192.09ms and the lower CI is 177.19ms for a 95% confidence level. The confidence interval at 95% produces a small bound for OTA update time and delay calculations. From these results, we conclude that our proposed approach is scalable and can offer the desired performance, even for a large number of vehicles.

Effect upon varying packet sizes and vehicles

To understand the impact of varying the number of vehicles on the software update process, we measure the average throughput after latency calculation. The throughput (DR_i) is the total amount of transferred data from fog nodes to any vehicle with respect to time. Equations 4.4 and 4.5 determine the throughput for sending different number of packets (Pkt^j_{size}) against varying traffic size.

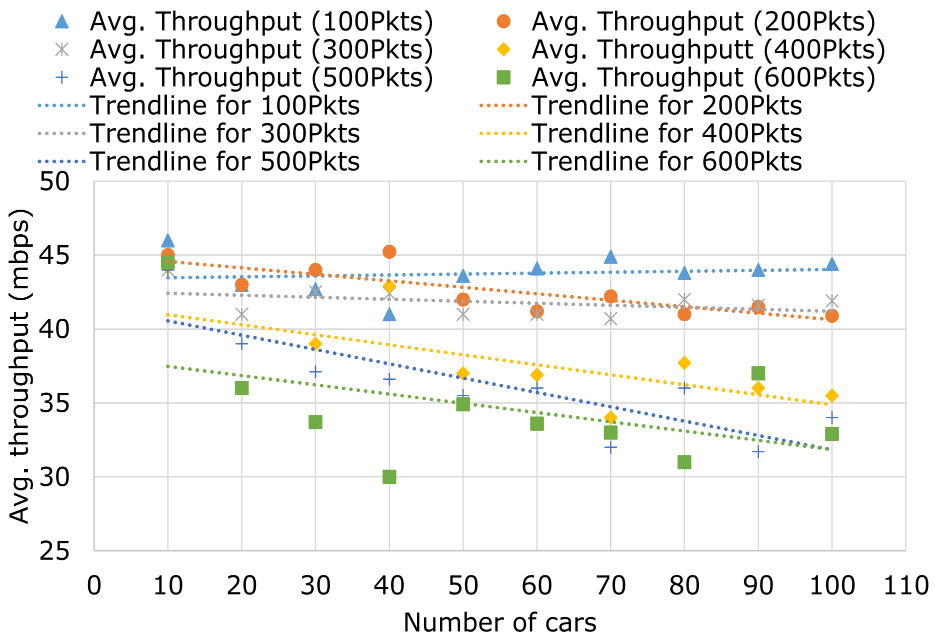


Figure 4.14: Effect on throughput for increasing number of vehicles and software sizes

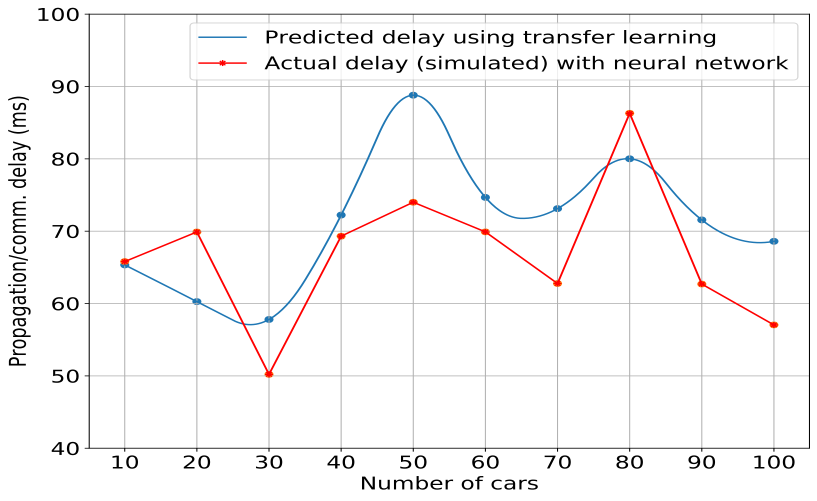


Figure 4.15: Actual communication delay comparison with predicted delay using Transfer Learning

Figure 4.14 show the average throughput against different software sizes for an increasing number of cars.

It is visible that the throughput changes while varying the number of vehicles. Moreover, we observe that the throughput is maximized in most cases when the traffic size (number of cars) remains small. For example, when the number of cars is 10, the average throughput for different software sizes is about 45 Mbps. The standard deviation among these throughputs is also small. However, when the number of cars increases, we see that the average throughputs for small software sizes are comparatively higher than the larger software sizes. For updating the software sizes of 500 and 600 packets, the average throughput is always below 40 Mbps due to larger resource demand with propagation delay. On the other hand, the trend line for 100 packets shows how the throughput changes with a small deviation for low resource utilization and low propagation delay. Similarly, the trend line for 800 packets has a larger deviation than the others.

Apart from this, we compare the predicted communication delay for an individual car with the actual communication delay (simulated), where the predicted communication delay is either close to the actual value or a little higher than the actual. Figure 4.15 shows the comparison between the predicted communication delay using transfer learning and the actual communication delay (simulated) with neural network.

4.3 ML-based Fog-Assisted Embedded Applications Development Framework

4.3.1 Resource-intensive ML-based Next-Gen Embedded Applications

The rapidly evolving landscapes of embedded systems and Cyber-physical Systems (CPS) have seen a significant upsurge in the incorporation of Deep Neural Networks (DNNs). Innovative advancements in fields like autonomous driving, robotics, unmanned aerial vehicles (UAVs), industrial automation, and the Internet of Things (IoT) owe their success to DNNs due to their unparalleled proficiency in handling intricate tasks with precision [126], [127].

However, with the demand for intelligent systems surging, the architecture of computing platforms has been forced to evolve, especially to accommodate the necessities of faster computations vital for real-time decision-making [128], [129]. Delegating the execution of DNNs to distant servers, such as those in cloud configurations, might introduce significant latency. This delay, primarily induced by the transmission of voluminous data over networks, often culminates in prolonged processing intervals. Practical scenarios, such as UAV operations in hostile environments, further accentuate these challenges due to factors like weather adversities affecting both communication and processing [130].

Herein, fog computing, a paradigmatic extension of cloud computing, offers a promising solution. By provisioning computational capabilities at the edge of the network, closer to data sources, fog computing effectively diminishes latency, enhancing application response times. This mode of computing encompasses edge computing, where the computational processes are shifted to edge devices. These devices, being proximate to the data sources, further reduce the latency, providing a more distributed computing framework.

Deploying DNN models, especially on edge devices that form an integral part of the fog computing framework, is not devoid of challenges. The constraints of computational power and memory on these devices often preclude the deployment of extensive DNN models. For instance, the large size and computational demands of Convolution Neural Networks (CNNs), a variant of DNNs, make them unsuitable for singular core edge devices [131]. The traditional approach, wherein machine learning-driven CPS applications operated sequentially on single-core processors or devices, is rapidly becoming obsolete [42]. The pressing requisites of advanced applications have necessitated the shift towards parallel computing.

4.3.2 Need for Parallelizing ML Model Training on Embedded Architectures

- *Resource Limitations:* Embedded systems often have limited computational and memory resources. Therefore, parallelizing the training process helps to reduce the overall computation.
- *Timing Constraints:* Training a large and complex DNN takes significant time, and embedded applications often have timing constraints. Parallelizing the process lowers the training time by partitioning the workload across multiple processors.
- *Complex Models:* As model size and complexity continue to grow, parallelizing the training on multiple processors can manage the complexities of the workload.
- *Improved Accuracy:* Training a DNN on large datasets can potentially improve the accuracy, compared to training on smaller datasets. Therefore, the parallel training architecture can facilitate achieving improved accuracy.

4.3.3 Challenges in Model Parallelism and Performance Metrics

Distributed parallel computing, particularly in the realm of DNNs, faces multiple challenges. At the forefront are architectural dependencies and communication latency, which critically shape the performance outcomes of models [40]. Overlooking these elements could precipitate unbalanced workloads and resource inefficiencies. The allure of model parallelism, with its promise of enhanced performance, is nonetheless fraught with its own set of complications. The task of partitioning and coordinating DNNs across a range of edge devices is a challenge. The constricted bandwidth and computational prowess of edge devices can further reduce the efficiency of parallelism.

In our pursuit to optimize performance on edge devices, this research underscores the need for efficient DNN model partitioning and pipelined model parallelism. These considerations are crucial for harnessing the full potential of edge devices within the fog computing framework.

Another pivotal metric in assessing the computational demands of DNNs is FLOPs, or floating-point operations. FLOPs enumerate the arithmetic operations (like additions, subtractions, multiplications, and divisions) a neural network undertakes during its training or inference phase. This metric is indispensable for gauging the computational footprint of a neural network and often serves as a yardstick for contrasting the efficiencies of diverse models.

To illustrate this we consider the AlexNet deep convolutional neural network [44]. The FLOPs needed to compute the output of each layer can be approximated by tallying the arithmetic operations undertaken by that layer. For instance, the inaugural convolutional layer of AlexNet comprises 96 filters, each spanning 11×11. The layer's input is a $227 \times 227 \times 3$ image, employing a stride of 4 and zero padding. The operations needed to generate the layer's output can be broken down as follows:

FLOPs: FLOPs (floating-point operations) is a measure of the number of arithmetic operations (such as additions, subtractions, multiplications, and divisions) performed by a neural network during training or inference. It is used to estimate the computational cost of a neural network and is often used as a metric for comparing the efficiency of different models. For example, in the case of AlexNet [44] deep convolutional neural network, we can estimate the FLOPs required to compute the output of each layer by counting the number of arithmetic operations performed by the layer. The first convolutional layer of AlexNet has 96 filters, each with a size of 11×11. The input to the layer is a $227 \times 227 \times 3$ image. The layer uses a stride of 4 and padding of size 0. To compute the output of the layer, we need to perform the following operations:

Calculation of FLOPs: We have to apply convolution operation to the input image using all filters. For example, Convolution layer one has 96 filters. For each filter, we perform 11×11 multiplications. Since the output size is $55 \times 55 \times 96$, the total number of operations:

$$\begin{aligned} \textit{FLOPs for Conv} &= \textit{Number of Kernel} \times \textit{Kernel Shape} \times \textit{Output Shape} \\ &= 96 \times (11 \times 11 \times 3) \times (55 \times 55) \approx 105\text{M}\ [conv1] \end{aligned}$$

$$\begin{aligned}\textit{FLOPs for Fully Connected Layer} &= \textit{Input Shape} \times \textit{Output Size}\\ &= (6 \times 6 \times 256) \times 4096 \approx 37\text{M } [FC6]\end{aligned}$$

- Number of Kernel: the number of filters in the first layer (96)
- Kernel Shape: the size of each filter (11 x 11 x 3)
- Output Shape: the size of the output tensor (55 x 55 x 96)

The total FLOPs for all AlexNet layers is approximately 725M [132], as shown in Table 4.4.

Table 4.4: FLOPs calculation for AlexNet

Layer	**Input Size**	**Filter**	**Output Size**	**FLOPs**
Conv1	227x227x3	11x11x3	55x55x96	105,705,600
Conv2	27x27x96	5x5x48	27x27x256	223,948,800
Conv3	13x13x256	3x3x256	13x13x384	149,520,384
Conv4	13x13x384	3x3x192	13x13x384	112,140,288
Conv5	13x13x384	3x3x192	13x13x256	74,760,192
FC6	6x6x256		4096	37,748,736
FC7	4096		4096	16,777,216
FC8	4096		1000	4,096,000
Total	-	-	-	**725,147,008**

The significance of FLOPs extends beyond just model evaluation. It serves as a predictor for estimating model training durations.

Estimating Training time from FLOPs:

- *Calculate the total number of iterations:* Assuming a batch size of 128 and a total of 1000 images, we can estimate the number of iterations required to complete one epoch of training. Specifically, we will have $1000/128 = 7.8125$ iterations per epoch. Since we train our model for 100 epochs, the required total number of iterations will be $7.8125 * 100 = 781.25$.
- *Calculate the total number of FLOPs:* As we calculated before, the total number of FLOPs required to process one image through the AlexNet model is approximately 725 million. Since we are using a batch size of 128, the total number of FLOPs per iteration is 725 million$*$ 128 $=$ 92.8 billion.

Therefore, the total number of FLOPs required for training is 92.8 billion $* 781.25 = 72.4$ trillion FLOPs.

- *Estimate the training time:* Given a Raspberry Pi 4 with 4GB RAM and 1.5 GHz quad-core ARM Cortex-A72 CPU, the peak performance is around 2.8 GFLOPS. Each core can perform two floating-point operations per cycle, leading to a theoretical 3.0 GFLOPs; however, considering real-world factors, we assume a 2.8 GFLOPs peak performance. We can estimate the training time by dividing the total number of FLOPs by the processing power of the device:

$$\begin{aligned} \textit{Training time} &= \textit{Total FLOPs/Device Performance} \\ &= \text{72.4 trillion/2.8 billion FLOPS} \approx \text{25,857 seconds} \end{aligned}$$

4.3.4 Problem Formulation and Optimization

Partitioning a DNN model M across a set of N edge devices, each with different computational capacities, is a challenge. The goal is to minimize the total execution time of the model. Mathematically, this can be expressed as:

$$\min_P \sum_{i=1}^{L} \sum_{k=1}^{N} x_{ik} \cdot T_{ik} \tag{4.12}$$

Where $P = \{x_{ik}\}$ is a set of binary variables representing the assignment of each layer i to an edge device k. This assignment must satisfy certain constraints:

Capacity constraint: The computational workload of each layer assigned to an edge device must not exceed the computational capacity of that device:

$$\sum_{i=1}^{L} F_i \cdot x_{ik} \leq C_k, \quad \forall k \in 1, 2, \ldots, N \tag{4.13}$$

Communication constraint: The communication time for each layer assigned to an edge device must not exceed the worst-case execution time of the layer on an edge device with minimum computational capacity C_{worst}:

$$\sum_{i=1}^{L} \Delta_{ik} \cdot x_{ik} \leq T_{worst}, \quad \forall i, k \tag{4.14}$$

Partitioning constraint: Each layer must be assigned to exactly one edge device:

$$\sum_{k=1}^{N} x_{ik} = 1, \quad \forall i \in 1, 2, \ldots, L \tag{4.15}$$

Here, F_i is the number of FLOPs of layer i in the DNN model M, Δ_{ij} is the communication overhead between layer i and edge device k, which includes both the communication costs associated with partitioning the DNN layers across edge devices and the communication costs incurred during the training process, such as exchanging weight updates and aggregating them among workers. Moreover, C_k is the computational capacity of the k-th edge device, D is the input data size, T_{ik} is the execution time of layer i on edge device k, T_{worst} is the worst-case execution time of layer i on an edge device with minimum computational capacity C_{worst} in the considered network, and L is the total number of layers in the DNN model. The objective function seeks to minimize the total execution time of the model across all edge devices by finding the optimal assignment of layers to edge devices, subject to capacity and communication constraints. The partitioning constraints ensure that each layer is assigned to exactly one edge device.

In our work, we utilize mixed-integer linear programming (MILP) solvers such as IBM CPLEX or Gurobi to handle this optimization challenge. These solvers efficiently handle binary variables, linear constraints, and integer constraints. We further enhance the solution process by introducing problem-specific heuristics. The outcome aims to optimize the DNN model's performance on edge devices with restricted computational resources and bandwidth.

4.3.5 System Architecture and Assumptions

Our proposed system model for DNN model partitioning focuses on pipelined model parallelism, utilizing a heterogeneous multiprocessor edge device platform tailored for DNN deployment. The primary aim is to optimally partition the DNN model into sub-models and allocate them among edge devices or multi-core processors, ensuring parallelism. We assume that a multiprocessor edge device is equipped with several cores designed for parallel computation, and it could either be a general-purpose or an embedded processor.

Using the AlexNet CNN architecture as a baseline, our methodology divides the CNN model into sub-modules, mapping them for parallel execution using a task graph model. The parallel execution of these sub-modules should generate the same output as running the program on a single processor. The design architecture for the proposed DNN model partitioning for cyber-physical or embedded applications is illustrated in Figure 4.16. It is assumed that DNN applications require faster model training to meet their task requirements, which is facilitated by N number of edge devices or processors. For executing partitioned sub-modules, a task graph, represented as a directed acyclic graph (DAG), captures computational tasks, communication costs, and dependencies between the DNN model layers. The proposed DNN model partitioning algorithm determines the optimal number of partitions for the DNN model while considering the available edge resources and their communication times. The tasks controller then analyzes the partitioned modules and schedules their execution in a pipeline fashion to reduce computation time.

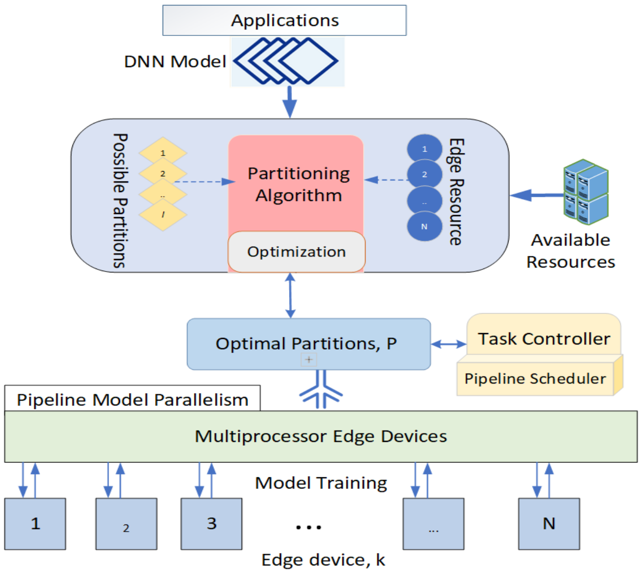


Figure 4.16: Design overview for DNN model execution on edge devices

4.3.6 Proposed Solutions for Model Partitioning and Pipeline Execution

In our work, we present a strategy for determining the number of partitions needed for the parallel execution of the Deep CNN model on edge devices. The partitioned modules are then allocated to the available edge devices, taking into account their computational capabilities and the inter-process communication latency. By integrating both data parallelism and model parallelism, our approach aims to achieve faster computation as opposed to the traditional sequential DNN model training.

Model Partitioning

Central to the concept of CNN model parallelism are its layers, showcased in Figure 4.17. These layers consist of convolutional and fully connected components, which can be spread across multiple processors to optimize model training.

However, dividing all the layers equally may result in increased communication overhead during training. Therefore, identifying the ideal number of partitions is a crucial aspect of deep learning model parallelism. We propose Algorithm 8 to determine the optimal partitions for running CNN layers on edge devices.

Algorithm 8: Finding Optimal DNN Model Partitions for Edge Devices

```
Input  : DNN model $M$, Number of partitions $S$, Set of edge devices $N$, Computational
         capacity $C_k$, Input data size $D$, Communication overhead $\Delta_{ik}$
Output: Optimal partitions of Model $M$ for edge devices
1  begin
2      Compute FLOPs $F_i$ of each layer in $M$
3      Compute total FLOPs $F_{\text{total}} = \sum_{i=1}^{L} F_i$
4      Compute the execution time $T_i$ for each layer $i$
5      Compute $T_{\text{worst}}$ for $M$ on a single edge device with capacity $C_{\text{worst}} = \min_k C_k$
6      Initialize optimal partitions $P^* = \{\}$ and $obj^* = \infty$
7      for $n = 1$ to $S$ do
8          Initialize binary variable $x_{i,k}$ for each layer $i$ and edge device $k$
9          Define objective function $obj_n$
10         Define capacity constraint for all $k \in N$
11         Define communication time for all $i$ and $k$
12         for $i = 1$ to $L$ do
13             for each $k$ in $\{1, 2\}$ within $N$ do
14                 Define communication constraint
15             end
16         end
17         Solve MILP problem to find $P_n$ and $obj_n$
18         if $obj_n < obj^*$ then
19             Set $P^* = P_n$ and $obj^* = obj_n$
20         end
21     end
22     return $P^*$
23 end
```

The proposed algorithm aims to optimize the training time of a deep neural network (DNN) model by partitioning it across a set of edge devices. The algorithm requires the DNN model, the number of edge devices, and the computational capacity and communication overhead of each device as input. Initially, the algorithm computes the FLOPs of each layer and the worst-case execution time of the model on a single-edge device with the minimum capacity. It then initializes the optimal partition and best objective value to empty and infinity, respectively.

Next, the algorithm sets up a mixed-integer linear programming (MILP) problem for each layer and edge device with binary variables. The objective function is set to minimize the total execution time of the model, taking into account the FLOPs of each layer and the time taken to execute the layer on each edge device. Capacity and communication constraints are added to prevent the computational workload and communication time from exceeding the available resources.

The algorithm then solves the MILP problem for each partition to determine the optimal partition and corresponding objective value. If the objective value is lower than the current best objective value, the optimal partition and best objective value are updated accordingly. The algorithm continues this process for each partition and returns the optimal partition with the minimum execution time. This approach can enhance the performance of DNN models on edge devices with limited computational resources and bandwidth.

An Illustrative Example

To illustrate our proposed partitioning algorithm, we provide an example of how to optimally partition the AlexNet DNN model across four edge devices with computational capacities of approximately 200M, 100M, 70M, and 50M FLOPs/s. We first calculate the FLOPs for each layer of the AlexNet model, as shown in Table 4.4, with a total of approximately 725M FLOPs for all layers. We then compute the execution time for each layer on each device using the FLOPs and the computational capacity of each device, as presented in Table 4.5. To determine the optimal partitions for the AlexNet DNN model, we continue with the following steps:

- We compute the worst-case execution time of AlexNet on a single edge device with a capacity of $C_{worst} = \min(C_k)$: $C_{worst} = 50,000,000$ FLOPs/s (capacity of Device 4), and T_{worst}=F_{total}/C_{worst} =14.503 s.
- For each partition n (in this case, we try with $S = 4$ possible partitions), we define the capacity and communication time (assuming $0.1s$) constraints.
- Finally, we run the MILP optimization, which yields the optimal partition P^* and the best objective

Table 4.5: Execution time of AlexNet layers on different edge devices

Layer	FLOPs	Device 1	Device 2	Device 3	Device 4
1	238,878,720	1.19 s	2.37 s	3.56 s	4.74 s
2	1,074,042,624	5.37 s	10.73 s	16.10 s	21.46 s
3	231,211,264	1.16 s	2.32 s	3.47 s	4.63 s
4	231,211,264	1.16 s	2.32 s	3.47 s	4.63 s
5	173,408,256	0.87 s	1.73 s	2.60 s	3.46 s
6	86,704,640	0.43 s	0.87 s	1.30 s	1.73 s
7	41,943,040	0.21 s	0.42 s	0.63 s	0.84 s
8	2,097,152	0.01 s	0.02 s	0.03 s	0.04 s

value obj^*. However, many factors, such as the choice of objective function and communication constraints, can affect the performance of the partitioning algorithm in practice.

Table 4.6: Optimal partitioning of AlexNet DNN model over four distinct edge devices

Partition Number	Layers	Devices	Execution Time
1	1, 2, 3, 4, 5	1	12.03 s
2	6, 7	2	0.137 s
3	8	3	0.007 s
		Total	**12.202 s**

The optimal partitioning result of the AlexNet model is presented in Table 4.6. The optimization algorithm has determined that the optimal partitioning of the AlexNet model is to divide it into three partitions, with the first partition consisting of layers 1 to 5, the second partition consisting of layers 6 and 7, and the third partition consisting of layer 8. This partitioning results in an overall execution time of 12.202 seconds, which is faster than the worst-case execution time on a single-edge device. This shows the advantage of using edge computing and optimal partitioning to speed up the execution of DNN models.

Pipelined Execution

According to the designed architecture shown in Figure 4.16, the task controller is responsible for distributing the partitioned modules of the CNN model to different processors based on edge resources. Algorithm 8 is responsible for finding the optimal partitions of the model, as shown in Figure 4.17 for the AlexNet model. This model consists of five consecutive convolutional layers and three fully connected layers. For better performance, data parallelism is more effective in training convolutional layers, while model parallelism is

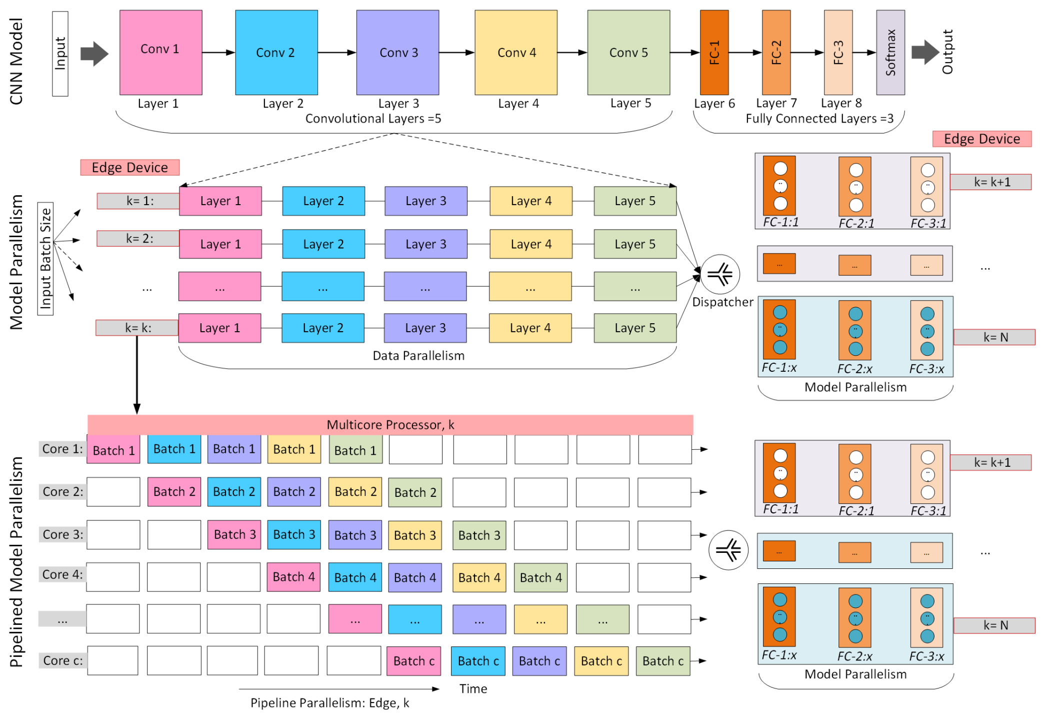


Figure 4.17: Pipelined model parallelism on edge devices

more suitable for dense or fully connected layers. The framework implements data parallelism by duplicating the convolution layers on computing processors to run input batch data in parallel. On the other hand, the fully connected layers are split into two parts, and model parallelism is applied to train them across the model dimension.

To achieve efficient data parallelism and optimize processor utilization when training the convolutional layers, we propose implementing a pipeline execution of mini-batch input data. Figure 4.17 illustrates this approach, where each processing core handles an input batch data and runs the partitioned model. For instance, if a multi-core edge device is assigned to execute a partitioned module containing all convolution layers, the first core sequentially runs the convolution layers using the first input batch data, while other processing cores begin computing subsequent input batches using the pipeline approach. As soon as the first core completes the execution of the last layer, the next available core immediately starts executing the next input batch.

Model Training on Edge Devices

The proposed approach in Figure 4.17 for DNN model training employs a dispatcher that sends input data batches to dense layers once convolution layers complete their training. Fully connected layer partitions receive the batch data for training the sub-module, and the dispatcher sends the input batch data sequentially to maintain the forward and backward propagation. During training, the number of workers, w, is determined based on the number of available processors. Each worker is assigned to an edge device with the corresponding partitioned portion of the model. Workers update weights using gradient descent, and the primary worker aggregates weights from all other workers. The performance of the optimal number of partitions is evaluated based on the communication latency and execution time of each layer, and memory usage is monitored for under-utilization or overloading.

4.3.7 Evaluation of Proposed Model Partitioning Framework

Environment Setup and Dataset: The proposed framework's performance was evaluated for image object detection on an embedded system functioning as an edge device. The experiments were conducted on various architectures, including AlexNet [44], ResNet [133], and VGG-16 [134], considering their relevance, computational complexity and popularity. Experiments were performed on the NVIDIA Jetson Nano [135], which is equipped with Quad-core ARM Cortex-A57 processors. The operating system used for the experiments was Ubuntu 20.04 LTS, and the Jetson Nano had 4 GB of 64-bit LPDDR4 memory. In this experiment, an image dataset [136] for plant leaf disease detection is utilized, consisting of over 50,000 images, each classified into one of 38 disease classes. The dataset is divided into training and testing sets, with 80% for training and 20% for testing, after resizing the images to 227×227.

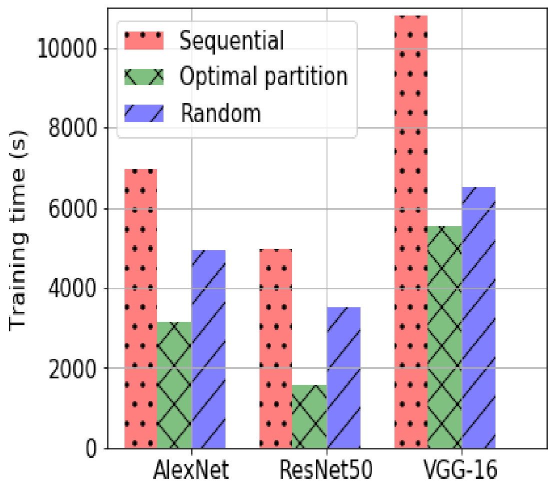
Efficiency Evaluation

The purpose of this experiment is to assess the effectiveness of different CNN architecture partitioning techniques for model parallelism, with the goal of reducing the model training time. The training is performed using the python multiprocessing library, Ray [137], on a Jetson Nano board with an SGD optimizer, for 20 epochs, where each epoch has 200 steps with a batch size of 32. The net execution time is measured for each epoch as the model is partitioned into varying numbers of cores. The experiment is conducted on a Jetson Nano board with four available processors. The proposed algorithm, Algorithm 8, is utilized to determine the optimal number of partitions for the AlexNet, ResNet50, and VGG-16 architectures. The maximum number of partitions is limited to four, due to the limited number of CPU cores, with the optimal number of partitions being found to be three, four, and four, respectively. The net execution times for the three architectures are analyzed for different numbers of partitions.

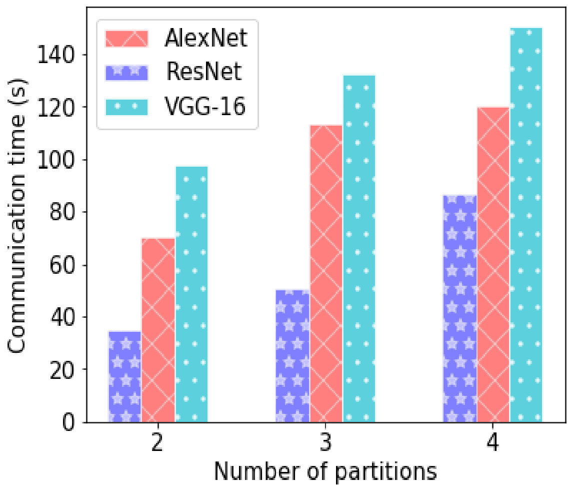
Table 4.7: Pipelined model parallelism performance for different CNN networks

Networks	**Layers** (L)	**FLOPs**	**Partitions** (S)	**Epochs**	**Pipelined Parallel Training Time**	**Sequential Training Time**	**Accuracy (Pipelined)**
AlexNet [44]	8	725M	3	20	3129.4s	6950.7s	96.1%
ResNet50 [133]	50	3.8G	4	20	1574.5s	4956.5s	97.6%
VGG-16 [134]	16	16G	4	20	5525.6s	10792.3s	90.3%

The optimized function given in Equation 4.12 was used to calculate the training time T_{ij} of each layer, as shown in Table 4.7. The results reveal that pipelined parallel computing for AlexNet, with three optimal splits, takes approximately 3129.4s, yielding a speed-up of 2.3 times over serial execution on a single core. Similarly, the ResNet50 and VGG-16 models show the lowest execution time when split into four CPU cores, with speed-ups of 3.2 times and 2.0 times, respectively, over sequential executions.



(a) Training time for partitioning model randomly



(b) Comm. time for different number of partitions

Figure 4.18: Comparison of training time and communication time for model partitioning

To evaluate the effectiveness of pipelined model parallelism for optimal partitioning, we compared the training time of the random partitioning approach, which partitions the model randomly. Figure 4.18(a) shows the comparison of training time among the non-partitioning (sequential), random, and optimal partitioning approaches. It is observed that the optimal partitioning approach minimizes the training times for all three CNN networks. Figure 4.18(b) illustrates the communication latency for different partitions over available edge devices or processors. The results indicate that the latency rises with the increasing number of model splits, particularly for large DNN models with a significant number of trainable parameters. VGG-16, with its high number of parameters, demonstrates the highest latency. The experiment demonstrates an average speed-up of 2.5 times with pipelined model parallelism over sequential executions. The potential of pipelined model parallelism to enhance the performance of DNN applications in multiprocessor edge devices is evident. We expect the acceleration to improve further with increasing training epochs for larger DNN models.

Threats to Validity

To ensure the scope and applicability of our work, we identify potential threats to the validity of our proposed approach.

Generalizability to all DNN applications: Our approach is focused on the CNN architecture, which is a specific type of DNN. Therefore, the effectiveness of our model-splitting approach may be limited to architectures that have a similar structure to CNN.

Applicability to GPU-enabled edge device: While GPU support is available, our work mainly focuses on supporting multiprocessor edge devices that predominantly use multi-core CPUs. This is to enable the reuse of existing embedded systems and avoid additional costs. However, our approach can be extended to support multi-GPU edge devices as well.

4.4 Safe Online ML Model Training and Inference

The prominence of edge networks in contemporary cyber-physical system (CPS) applications, such as autonomous driving systems, robotics, and health monitoring, has expanded rapidly [138]. These applications heavily leverage real-time machine learning (ML) decision-making capabilities, resulting in edge networks becoming instrumental in facilitating continuous adaptation of ML algorithms like SVM, RF, and DNNs in CPS applications [139]. However, online model training, particularly in an edge environment, introduces substantial challenges related to model parallelization, safety considerations, and more [91]. Addressing these challenges is vital for enhancing ML model performance on edge networks and ensuring system reliability.

4.4.1 Current Landscape and Limitations

Although various studies have delved into optimizing ML model training on edge networks [27], many of these solutions focus on data and model parallelism, overlooking safety and reliability issues intrinsic to ML model partitioning. While the works of Wen Sun *et al.* [140], Guangxu *et al.* [141], and Sina *et al.* [142] emphasized reducing training time and offloading, the safety and reliability of online model training were less emphasized. An approach that is comprehensive in its design, optimizing model splits, and adhering to safety standards like IEC 61508, ISO 26262, and UL 4600 is indispensable.

4.4.2 Safety Constraints Integration for Safe Online Model Training

Our primary goal is to devise an optimal method for ML model partitioning across edge devices, balancing training time, communication latency, and TMR time. The optimization problem can be written as:

$$\min_{s_k \in S, \forall k} \quad \sum_{m_i \in s_k} t_{m_i} + \beta \sum_{m_i \in s_k} l_{m_i} + \gamma \sum_{k=1}^{3} T_{TMR_{k,m_i}} \tag{4.16}$$

The equation 4.16 has three main components. The first term in the optimization problem represents the total training time for all splits assigned to edge devices in each partitioning strategy. The second term denotes the total communication latency for all splits assigned to edge devices in every partition, capturing the costs associated with transmitting data between devices. The third term corresponds to the total time spent on TMR, which ensures the system's reliability by incorporating redundancy into the distributed ML model. We introduce two weighting factors, β and γ, to balance the trade-offs between these components:

- β: A weighting factor that balances the importance of communication latency in the optimization problem. A higher value of β emphasizes minimizing latency, whereas a lower value focuses more on minimizing training time.
- γ: A weighting factor that balances the importance of TMR time in the optimization problem. A higher value of γ emphasizes minimizing TMR time, whereas a lower value focuses on balancing training time and communication latency.

Constraints

- **Model Partition Constraint:** This constraint ensures that the ML model is properly partitioned across the edge devices.

$$\bigcup_{i=1}^{m} s_{k,i} = M; \quad \forall k = 1, \ldots, |S| \tag{CT1}$$

Constraint (CT1) ensures that the union of all model splits across all partitions and edge devices covers the entire set of model splits, which is equal to the full model M.

$$S = \{s_1, s_2, \ldots, s_n\} : s_k \in S, k = 1, \ldots, n \tag{CT2}$$

Here, s_k represents the k-th partition assigned to different edge devices. The set S contains all possible assignments of partitions to edge devices. For our example with 6 model splits (m_1, m_2, m_3, m_4, m_5,m_6) and 3 edge devices, the set S could contain possible partitions like:

$$\begin{aligned} S = \{&(\{m_{1,1}, m_{2,1}\}, \{m_{3,2}, m_{4,2}\}, \{m_{5,3}, m_{6,3}\}),\\ &(\{m_{1,1}, m_{2,1}, m_{3,1}\}, \{m_{4,2}\}, \{m_{5,3}, m_{6,3}\}),\\ &\vdots,\\ &(\{m_{1,1}, m_{2,1}\}, \{m_{3,2}, m_{4,2}, m_{5,2}\}, \{m_{6,3}\})\} \end{aligned}$$

So, tuple $s_k = (s_{k,1}, s_{k,2}, ..., s_{k,e})$, where e is the number of edge devices, and $s_{k,i}$ is the set of model splits assigned to the i-th edge device in the k-th partition. The optimization problem aims to find the best assignment of partitions ($s_k \in S$) to minimize the total training time and communication latency while considering TMR.

- **Processing Capability Constraints:**

  These constraints ensure that the requirements of assigned model splits (processing, memory, and bandwidth) do not exceed each edge device's capabilities.

$$\sum_{m \in s_{k,i}} w_m \leq p_i; \quad \forall k = 1, \ldots, |S|; \quad i \in n \tag{CT3}$$

Constraint (CT3) ensures that the processing requirements (w_m) of assigned model splits do not exceed a device's processing power (p_i).

$$\sum_{m \in s_{k,i}} R_m \leq R_{M_i}; \quad \forall k = 1, \ldots, |S|; \quad i \in n \tag{CT4}$$

$$\sum_{m \in s_{k,i}} b_m \leq b_{B_i}; \quad \forall k = 1, \ldots, |S|; \quad i \in n \tag{CT5}$$

Constraint (CT4) guarantees that memory requirements (R_m) of assigned model splits do not exceed a device's memory capacity (R_{M_i}). Similarly, Constraint (CT5) makes sure that the bandwidth requirements (b_m) for transmitting assigned model splits do not exceed a device's bandwidth capacity (b_{B_i}).

- **TMR and Safety Constraint:**

$$\sum_{k=1}^{e} x_{k,j,m_i} = 1; \quad j = 1, \ldots, 3 \tag{CT6}$$

In this constraint, x_{k,j,m_i} represents the binary decision variable for the m_i-th model split assigned to the k-th edge device in the j-th TMR instance. The constraint ensures that exactly one edge device is assigned to each TMR instance for the m_i-th model split, which is important for ensuring the reliability of the TMR configuration and preventing errors or failures, contributing to the system's overall safety by minimizing the likelihood of incorrect or damaging outputs.

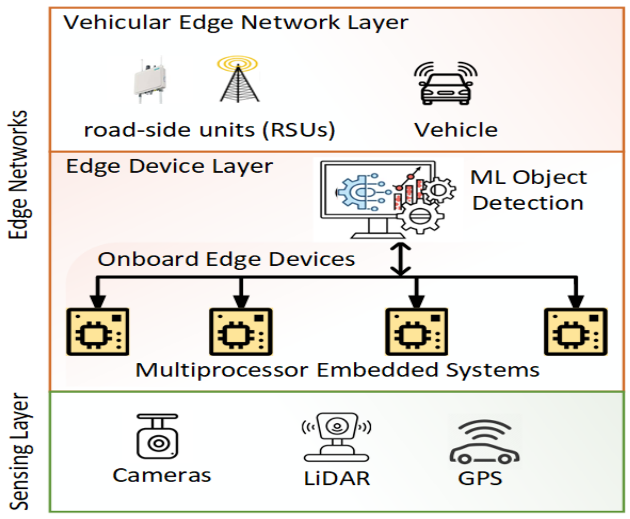


Figure 4.19: Machine learning-enabled edge networks

$$\sum_{k=1}^{e} x_{k,j,m_i} T_{TMR_{k,m_i}} \leq T_{fail,m_i} \tag{CT7}$$

The constraint CT7 ensures that the total time for training the redundant models and performing TMR in each TMR instance for the m_i-th model split does not exceed the specified failure threshold T_{fail,m_i}, determined by system designers or domain experts.

4.4.3 Proposed Safe ML Model Training Framework for Edge Networks

We introduce a robust framework designed to partition ML models for online training on edge networks. This framework, mindful of safety constraints and requirements, aspires to balance training time, communication latency, and resource utilization, ensuring reliable model updates on edge devices. Central to our approach is an intelligent partitioning algorithm that divides ML models into sub-models suitable for parallel execution across multiple edge devices. To further guarantee safety and reliability during online model training, the Triple Modular Redundancy (TMR) technique, a recognized Single Event Upset (SEU) solution, has been incorporated [143]. By leveraging TMR, our algorithm ensures system integrity even when faced with potential faults or hardware compromises.

Our proposed system is modelled around an autonomous vehicle system, which requires real-time ML

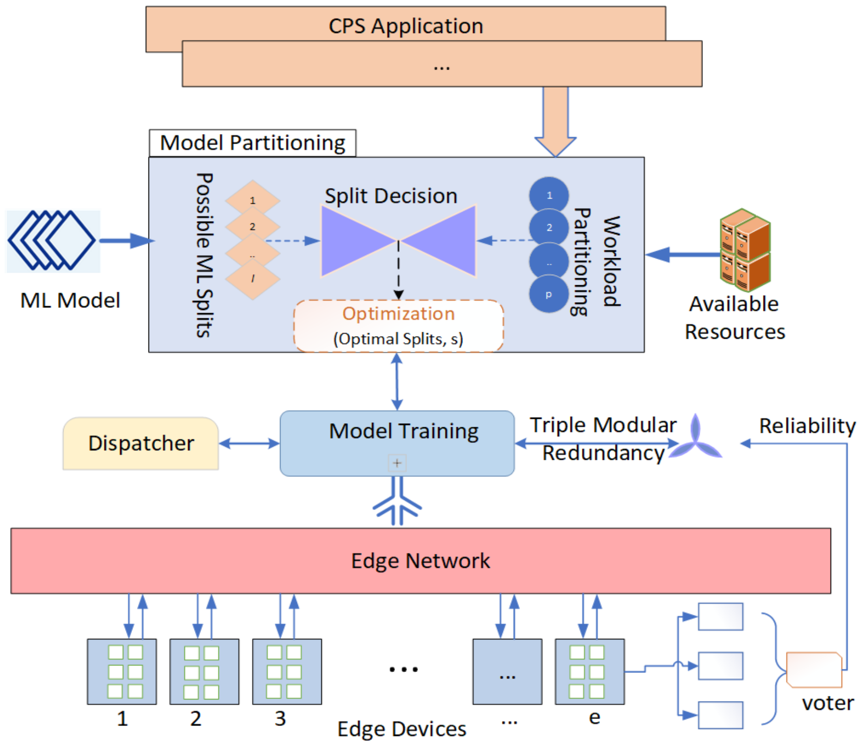


Figure 4.20: Proposed framework for ML model partitioning

model training for decision-making applications such as object detection. The system architecture, comprised of edge devices and a centralized server for model management, is depicted in Figure 4.19.

Framework Workflow and Methodology

Building on the aforementioned system, our framework presents a parallel computing architecture tailored for edge networks. Figure 4.20 outlines the detailed workflow of this proposed framework. Here, an optimal decision is made by minimizing the net training time, emphasizing a split decision module. This module employs a partitioning algorithm, determining not only the model splits but also their optimal execution environment, further enhancing application performance.

Model Partitioning Given the computational constraints of edge devices, we employ Algorithm 9 to discern the optimal partitioning s^* and map model splits to edge devices. This algorithm is particularly suited for models like SVM and RF, which we leverage in this study.

Algorithm 9: Optimal Model Partitioning and Mapping for Edge Networks

```
Input: ML model $M$, edge devices processing capability $p_i$ for device $d_i$, $\forall i \in n$, $\beta$, $\gamma$,
       communication latency $l_{m_i}$, training data
Output: Optimal partitioning $s^*$ and mapping of model splits to edge devices
   /* Initialize search space randomly */
1  $S \leftarrow s_1, s_2, \ldots, s_k$
2  while $\Delta f(s_k) \geq \epsilon$ do
       /* Evaluate the obj function for each $s_k$ */
3      for each partitioning $s_k \in S$ do
4          $f(s_k) \leftarrow$ evaluateObjectiveFunction$(s_k)$ based on Equation (4.16)
5      end
       /* Update the search space */
6      $S \leftarrow S \pm \Delta s_k$ to improve $f(s_k)$
7      Sort edge devices: sort$(P, p_i)$
       /* Assign model splits to edge devices */
8      for each submodel $m$ do
9          $p_{\text{min}} \leftarrow \arg\min_{p_i \in P} p_i$ s.t. (CT3)
           /* Update the available capacity */
10         $p_{\text{min}} \leftarrow p_{\text{min}} - w_m$
11     end
12     Calculate $l_{m_i}$ for each model split $m_i$
13     Sort model splits based on $l_{m_i}$: sort$(M, l_{m_i})$
       /* Schedule $m_i \in m$ on edge & update time */
14     for each model split $m_i$ do
15         $d_j$: $d_j \leftarrow d_j \cup m_i$; s.t. to (CT6) and (CT7)
16         $t_{m_i} \leftarrow$ calculateTrainingTime$(m_i, d_j)$
17         $TMR_{m_i} \leftarrow$ calculateTMRTime$(m_i)$
18     end
19     Calculate $\Delta f(s_k)$ = $f(s_k^{\text{old}})$ - $f(s_k^{\text{new}})$
20 end
   /* Select the best partitioning */
21 Select $s^*$ = $\arg\min_{s_k \in S} f(s_k)$
22 Train split $m_i$ on edge device with training data
23 For each model split $m_i$, perform TMR: $y_{m_i} \leftarrow$ majority$(y^1_{m_i}, y^2_{m_i}, y^3_{m_i})$
24 return $s^*$, mapping of model splits to edge devices
```

Algorithm Overview The proposed algorithm iteratively explores the solution space S with a set of random model partitionings $s_1, s_2, \ldots, s_k$. It evaluates each partitioning based on an objective function and updates the search space to improve the function value. The search process continues until the change in the objective function $\Delta f(s_k)$ is below a threshold ϵ. For each partitioning $s_k \in S$, the algorithm evaluates the objective function $f(s_k)$ using Equation (4.16) and updates the search space accordingly.

Edge Device Mapping: Post identifying the optimal partitioning, the algorithm maps each model split to an edge device based on its processing power. It sorts edge devices by processing capacity (sort(P, p_i)) and assigns each submodel m to the edge device with the least available capacity (p_{min}) in line with Constraint (CT3), updating the device's available capacity.

Scheduling and TMR Constraints: The algorithm schedules model splits for training on edge devices, considering communication latency l_{m_i} and TMR constraints. Model splits, ordered by their communication latency, are scheduled on assigned edge devices (d_j), optimizing training time while adhering to TMR constraints. Training and TMR times for each model split m_i on edge device d_j are computed and updated.

Training and TMR Integration: After selecting the optimal partitioning s^*, the framework conducts online training for each model split m_i on assigned edge devices, enabling real-time adaptation. Concurrently, it implements TMR for each m_i by selecting the output with at least two matching instances. The framework then returns the optimal partitioning s^* and model-to-device mapping. This optimal partition minimizes training time while ensuring worst-case execution or failure threshold T_{fail} is not exceeded, and the processor utilization remains within acceptable limits. This problem is solved using mixed integer linear programming (MILP), allowing continuous model improvement in a dynamic context.

Time Complexity Analysis of Algorithm 9: The time complexity of Algorithm 9 is driven by its key operations. The initialization of the search space S takes constant time, $O(1)$. The main loop, iterating over the total number of possible partitions, n_S, until the objective function change ($\Delta f(s_k)$) is below a threshold ϵ, has time complexity $O(E \cdot n_S)$ for the evaluation of the objective function and $O(U \cdot n_S)$ for updating the search space, where E is the time taken by objective function evaluation and U represents the time consumed in updating the search space. Sorting and assignment of edge devices and model splits result in complexities of $O(e \log e)$ and $O(n_m \cdot e)$ respectively, where e is the number of edge devices and n_m is the number of model splits. Finally, the calculation of latency and scheduling of model splits add complexities $O(n_m \log n_m)$ and $O(n_m)$, respectively. The overall time complexity can be approximated as $O(E \cdot n_S + U \cdot n_S + e \log e + n_m \cdot e + n_m \log n_m + n_m)$. This explains the algorithm's scalability and efficiency with larger datasets and complex partitioning scenarios.

SVM on Edge Devices: SVMs can be effectively deployed on edge devices by partitioning the training

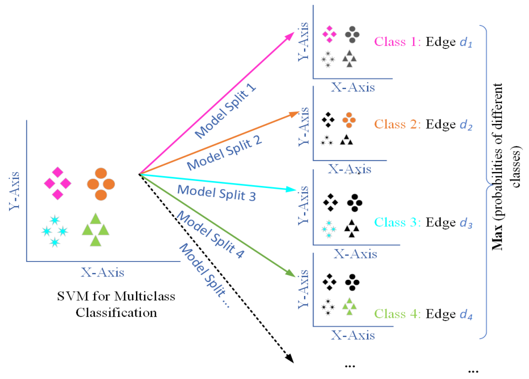


Figure 4.21: SVM model partition for parallel computing

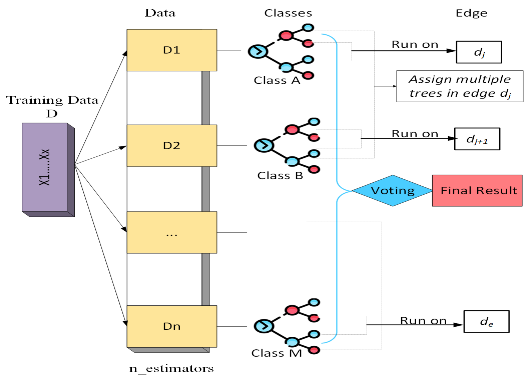


Figure 4.22: RF model partition for parallel computing

dataset and training multiple binary classifiers in parallel. For multi-class classification, one-vs-one or one-vs-rest approaches can be adopted, each binary classifier distinguishing between class pairs or one class against the rest, respectively. The partitioned classifiers can be assigned to different edge devices. After individual classifier training, outputs are combined using majority voting or other ensemble techniques to determine the final class label. This parallel structure, illustrated in Figure 4.21, allows for scalable, efficient SVM deployment on edge devices.

RF on Edge Devices: Training RF models on edge devices involves dividing decision trees and allocating them to different devices d_j. This method reduces the overall training time by leveraging the combined processing power. After training, outputs from individual trees are combined for the final prediction, as shown in Figure 4.22.

Dispatching Partitioned Models

This step assigns partitioned models to edge devices, maintaining execution order via associated threads. We use a global queue to join all processes and return the trained models to the master edge device. The master device combines all models to predict the training input. During partitioned model training, the master device ensures the correct integration of all processes.

Safe Integration using TMR

Safe execution is essential when partitioning models across edge devices. To counter issues like resource unavailability, aging, hardware compromise, data corruption, and side-channel attacks that may cause edge devices to produce incorrect results, we propose the integration of the TMR technique. This proven method enhances system reliability in edge network machine learning training. TMR, implemented in parallel on three devices, uses a majority voting system to eliminate single failure points. To ensure trusted computation, we calculate the training module's reliability. The reliability for training a partitioned model m_i at time t can be calculated using Equation 4.17, where R is the reliability of correct execution.

$$\begin{aligned} R(t) &= R^3(m_i,t) + 3(1-R(m_i,t)) * R^2(m_i,t) \\ R(t) &= 3R^2(m_i,t) - 2R^3(m_i,t) \end{aligned} \tag{4.17}$$

In the above Equation 4.17, $R^3(m_i,t)$ represents the probability that all three edge devices produce the correct output, and $3(1-R(m_i,t)) * R^2(m_i,t)$ is the probability that two out of the three edge devices produce the correct output while one fails. The final equation simplifies this computation.

4.4.4 Evaluation of the Proposed Safe Online Model Training

To evaluate the effectiveness of our proposed framework, we conduct different experiments in parallel computing architecture for multi-class classification problems using SVM and RF. We investigate the performance of the model partitioning algorithm for minimizing the net training time.

Dataset: The dataset employed in this study, titled "Traffic, Driving Style, and Road Surface Condition" [144] was sourced from Kaggle and initially used by Ruta *et al.* [145] to develop machine learning models for Internet of Things (IoT) applications. The data, comprising 24,957 data points, were collected from two vehicles, a Peugeot 207 1.4 HDi, and an Opel Corsa 1.3 HDi, using an OBD device paired with a smartphone. The dataset encompasses 14 features such as `altitude change`, `engine load`, `speed variance`, `fuel consumption`, etc. A comprehensive feature list and descriptions are available at [145]. The dataset is categorized into three sub-problems: road surface conditions, road traffic conditions, and driving style.

Environment: The experiments are conducted in multiprocessor systems, representative of advanced

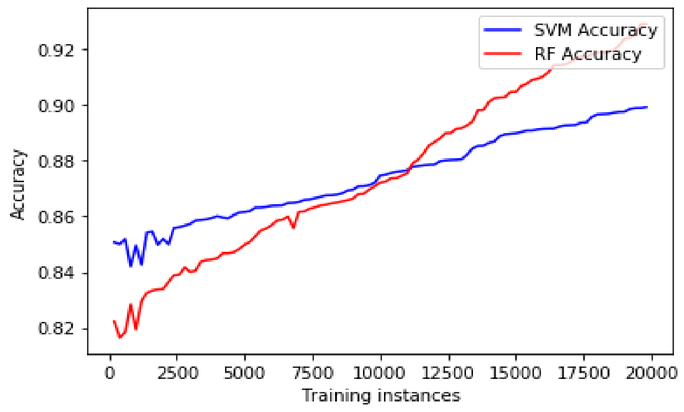


Figure 4.23: Training accuracy for online model training with input size 200

edge networks, with a shared network and a Linux 5.1.0-52-generic (x86_64) kernel integrated into the Ubuntu 20.04 LTS operating system. The multiprocessor systems consist of four Xeon(R) CPU E5-2623 v4 @ 2.60GHz processors, each endowed with eight cores, 4 GB of RAM, a 256 KiB L1 cache, and a 2 MiB L2 cache. This configuration provides a total of 16 GB RAM across the system, accommodating the SVM and RF models employed in our study. Additionally, the shared network offers sufficient bandwidth to meet the data flow requirements of our setup.

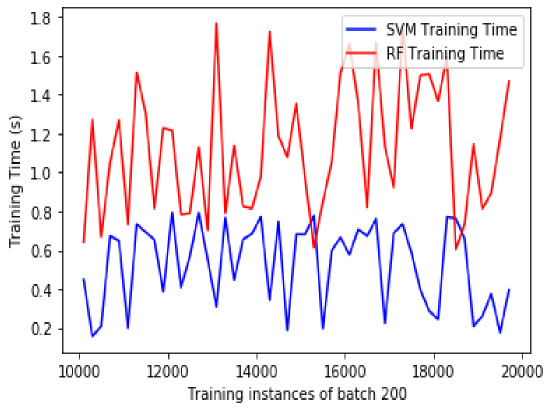


Figure 4.24: Training time for different training instances

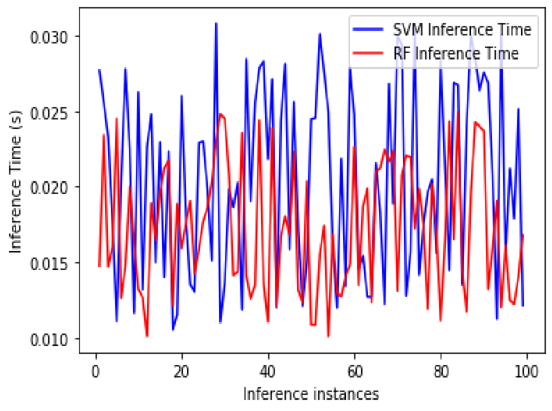


Figure 4.25: Inference time for different training instances

We opted for multiprocessor systems as an edge network to emulate modern edge devices' multicore structure. This allows us to explore parallel processing, resource allocation, and crucial inter-processor communication. This choice makes our study reflective of current edge capabilities and ensures relevance for real-world edge computing scenarios.

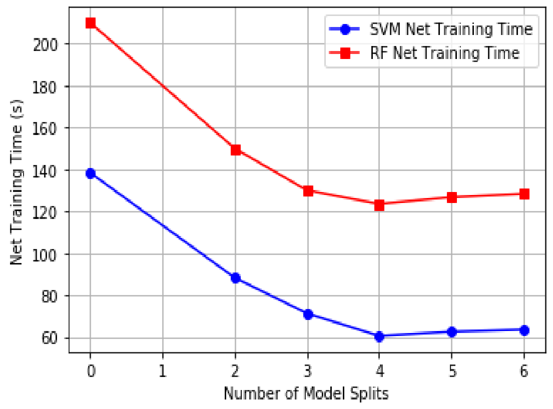


Figure 4.26: Training time for different number of model splits

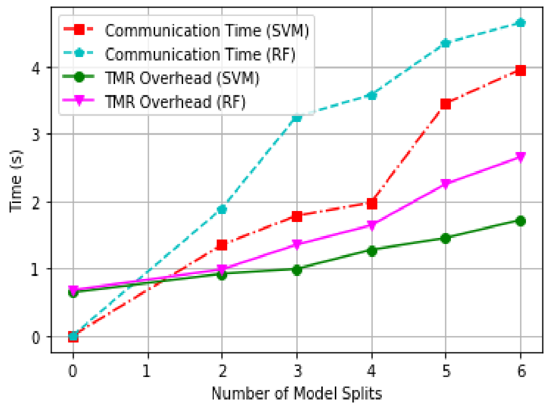


Figure 4.27: Comm. time and TMR overhead for different numbers of splits

Analysis of ML Model Parallel Computing on Edge

In this experiment, we used a pre-trained model as the basis for online model training. This initial model was trained on a dataset of 10,000 samples, which captured the general characteristics of the problem. For the online training phase, we utilized a batch size of 200 for partial model training, applying a learning rate of 0.001. The Algorithm 9 determined the optimal model splits by considering the communication latency and processing capability of each processor in the multiprocessor system. This approach enabled the efficient distribution of the model into four partitions across two edge devices or processors. For the training and testing datasets, the input feature shapes were (19965, 14) and (4992, 14). The online model training process incrementally updated the model using the remaining 9,965 samples beyond the initial 10,000 samples, allowing the model to adapt to new data patterns over time.

Accuracy The SVM and RF algorithms are evaluated based on their training accuracy. From Figure 4.23, it can be observed that the SVM model has a stable performance over the RF model. The SVM accuracies range from approximately 85% to 90%, while the RF accuracies range from around 82% to 93%. The performance improvement in the SVM model can be attributed to its ability to find an optimal hyperplane that maximizes the margin between different classes, making it well-suited for high-dimensional datasets like the one used in this experiment.

The learning rate (0.001) in this experiment impacts model training by modulating weight updates. Lower learning rates promote thorough solution space exploration, potentially increasing model accuracy despite slower convergence. Conversely, higher rates may lead to faster convergence, risking model accuracy due to the potential overshooting of the optimal solution. Our chosen rate of 0.001 seems to strike a balance between fast convergence and accuracy.

Training and Inference Time

We analyze the online model training time for both SVM and RF algorithms, which are used to process a batch size of 200 for the remaining 9,965 training samples on top of the pre-trained model. The training time for SVM and RF exhibits different trends, as shown in Figure 4.24. For the SVM, the training process utilizes the One vs. All (OVA) approach for multi-class classification. This enables the training of multiple binary classifiers in parallel, which results in reduced training time. On the other hand, the RF algorithm constructs an ensemble of 100 random decision trees, which can be computed in parallel by distributing the trees evenly among the available processors. The optimization algorithm finds the optimal distribution of the trees in a *for loop* parallelization, minimizing the overall training time.

Both classifiers are implemented using the $sklearn$ library with default hyperparameters, ensuring consistent configurations across the models. During the training process, the model split is executed on a separate processor, and the inter-processor communication time is recorded to assess the impact of parallel computation. We observe that the SVM generally takes less time in training than the RF. This difference can be attributed to the efficiency of the OVA approach in SVM training, which allows for faster parallelization and computation compared to the RF's decision tree construction and aggregation process.

Furthermore, we examine the inference times for 100 sample test inputs using the trained SVM and RF models. The inference times for both SVM and RF models are illustrated in Figure 4.25. It is apparent that both SVM and RF algorithms typically exhibit lower inference times. However, the RF model tends to have faster inference times compared to SVM. The maximum inference time for SVM is approximately 30.5 ms, whereas RF has an inference time of about 24.9 ms. In real-world CPS applications, inference time is crucial in determining the system's responsiveness, particularly when real-time decision-making is essential.

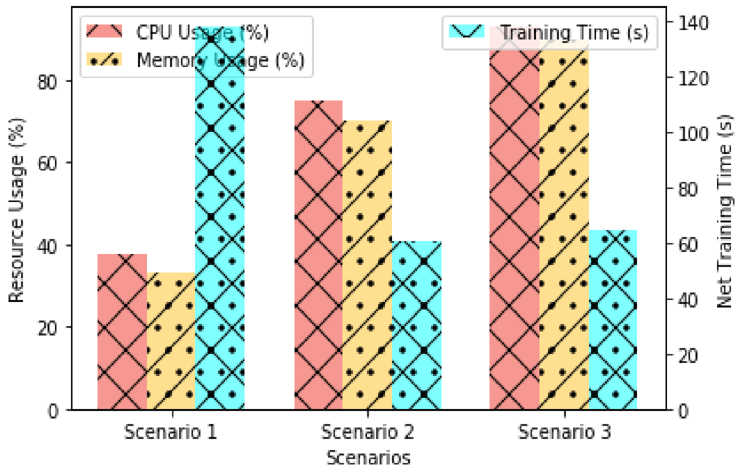


Figure 4.28: CPU and memory usage (%) for SVM model training

Comparing Net Training Time and Communication Latency

Net Training Time In our experiment, we analyzed the net training time for different numbers of model splits using both SVM and RF. We considered five different scenarios: no split (0), 2 splits, 3 splits, 4 splits, 5 splits, and 6 splits. For the no-split case (model split zero), the model is trained as a single unit without any partitioning. From split 2 onwards, the model is divided into the respective number of parts.

As illustrated in Figure 4.26, training time generally decreases with more splits, with exceptions noted for SVM at 5 and 6 splits. This indicates that the workload distribution across sub-models reduces training time efficiently up to 4 splits. Beyond this, no significant improvements were observed, while additional communication latency was incurred. The optimal split number for both SVM and RF was found to be 4, resulting in net training times of 60.48 and 123.5 seconds, respectively.

Communication Latency and TMR Overhead As shown in Figure 4.27, communication latency and TMR overhead were analyzed for various model splits. Communication latency tends to increase with the number of splits. This increase in communication latency can negatively affect the optimal number of model partitioning, as it can offset the benefits of reduced training time achieved through parallelization.

For the optimal number of splits (in this case, 4), SVM and RF models exhibited total latencies of 1.9784 and 3.58 seconds, respectively. Corresponding TMR overheads were 1.2785 and 1.6424 seconds. By integrating TMR into model training and operating devices in parallel, we efficiently ensure reliability, without significant delays in model training.

Resource Utilization Comparison:

In this experiment, we compare resource utilization across three SVM model training scenarios. Each considers different splits, and we measure CPU usage, memory usage, and training time with online training data (see Figure 4.28).

- **Scenario 1:** Single-edge device training (no split)
- **Scenario 2:** Training with two edge devices (4 splits)
- **Scenario 3:** Training with three edge devices (6 splits)

Scenario 1: Single-edge training (no split) Training is conducted on a single-edge device, utilizing one core. The remaining cores handle TMR configurations. CPU usage is 37.5% (3 cores out of 8), and memory usage is 33% for 4GB. As only one core is active, training takes longer (138.35 seconds), which may not always be efficient for real-time applications. This delay could impede real-time applications, highlighting the need for more efficient multi-core processing solutions.

cenario 2: Training with two edge devices (4 splits)

In this scenario, the model is divided into four and trained on two devices. Each edge device uses two of its cores to run two model splits in parallel and allocates the other cores for TMR.

As the number of utilized cores increases, the CPU usage rises from 37.5% to 75%, and memory usage rises to 70.2% for 8GB. However, due to parallel processing, training time reduces to 60.48 seconds, showing a 56.3% improvement over Scenario 1. This approach allows for more efficient and suitable training for real-time applications.

Scenario 3: Training with three edge devices (6 splits)

With six model parts assigned across three devices, more cores are used, leading to 93.2% CPU usage and 90% memory usage. Training time reduces further to 64.48 seconds but slightly longer than Scenario 2. This indicates that while more edge devices and model splits can increase resource usage, training time improvement may not necessarily scale proportionally.

However, optimal balance in device count, model splits, and resource utilization is key for efficient training in real-time applications. Scenario 2 appears to be the most efficient, but the optimal configuration will depend on specific application needs and edge device resources.

4.5 Summary

Enabling Fog Computing Architecture for Next-Gen Embedded Designs: Fog computing holds promise for the timely execution of latency-sensitive tasks. Using fog computing for faster OTA software updates in smart vehicles throws open some interesting challenges, such as handling handover, efficient fog node assignment for OTA updates, predicting communication delays and network traffic. Using k-means clustering, we offer a technique for distributing fog nodes post-traffic pattern analysis. With the predicted traffic load, we are able to enable or disable certain OTA updates delivering fog nodes in a particular region. This leads to the maximization of the net fog resource reserves and better utilization of such fog resources. We employ transfer learning that predicts the communication delay with less training time. This delay prediction helps evaluate the OTA update time sooner and more accurately. We demonstrate how to calculate the OTA update time in a cluster of vehicles. The effectiveness of the proposed approach is shown through the measured OTA update time, which confirms an improvement compared to the worst-case scenario. Moreover, we demonstrate the scalability of our proposed approach by showing how the system throughput changes for varying traffic and software size.

ML-based Fog-Assisted Software Development for Next-Gen Embedded Systems: In this section, we presented a new approach for the partitioning and pipelined execution of DNN models, specifically the CNN architecture, on edge devices. We demonstrated how the AlexNet DNN model can be optimally partitioned across edge devices with different computational capacities through an illustrative example. The pipelined execution strategy was discussed, detailing how each processing core handles input batch data and optimizes the execution of convolutional layers. We also addressed the training of DNN models on edge devices, emphasizing the importance of weight updates and gradient descent in a distributed setting.

Safe ML Training and Inference in Next-Gen Embedded Systems: Training machine learning models on edge networks are gaining traction, especially in the realm of real-time applications. Edge networks, with their inherent benefits of privacy, reduced overhead, and rapid decision-making capabilities, pose as a promising solution for many applications. However, their universal suitability is still under scrutiny. For real-time applications that prioritize fast responses, such as drones or autonomous vehicles, edge networks emerge as a potent contender. Nevertheless, the decision to employ edge networks should be intricately tied to the application's specific needs. One critical determinant of online model training's success on edge networks is the learning rate. This parameter orchestrates various aspects, including resource allocation, model adaptability, speed of convergence, and the overarching performance of the model. The art of fine-tuning the learning rate is, hence, paramount. Future research trajectories are poised toward its optimization for edge-based training.

In our exploration, we spotlighted the SVM and RF models, cherry-picking them for their distinctive learning algorithms and ubiquitous applications. With their modest computational requirements, these models made for an apt starting point, especially when considering devices with limited computational resources. However, these findings serve as a springboard. Their scalability to more intricate architectures, such as DNNs, is an imminent research challenge, paving the way for a broader assessment of the proposed approach's performance and applicability. Safety and reliability in online training are paramount, especially in real-time applications. The integration of the TMR technique in the proposed framework is a testament to this commitment. TMR ensures trusted computation by providing redundancy, thus safeguarding against potential faults or discrepancies during online training. This added layer of protection guarantees that even if one part of the system encounters an issue, the majority rule of TMR can identify and rectify it, ensuring safe and reliable model training. As ML models expand in intricacy and computational demands, the allure of parallel computing architectures in edge networks becomes undeniable. Our proposed framework, designed with optimal model partitioning and the safety net of the TMR technique, emphasizes the future of reliable and efficient ML model training on edge networks.

Chapter 5

FEATURE-BASED NEXT-GENERATION EMBEDDED SOFTWARE DEVELOPMENT

5.1 Introduction

Embedded systems, being integral components of Cyber-Physical Systems (CPS) and the Internet of Things (IoT), are increasingly shaping our interconnected world, with significant impacts on sectors like autonomous vehicles, smart cities, and precision agriculture [146]. These systems are at the forefront of technology, yet they are not without challenges.

The increased reliance on embedded systems has underscored the importance of robust and adaptable software development practices. One of the primary challenges arises from the need to integrate next-generation embedded applications and their computing paradigms. The current state-of-the-art [147], [148] struggles to fully leverage advanced applications, resulting in rigid and difficult-to-adapt or reuse systems. While these applications and paradigms promise unprecedented capabilities, they also introduce complexities. The inherent complexity and rigidity in many of the current systems often inhibit their adaptability to evolving requirements. Additionally, they complicate effectively managing feature dependencies, versioning, customization, and configuration, especially in distributed settings [149].

This chapter introduces a new perspective on next-generation applications by viewing them through the lens of feature-based software development. Instead of merely seeing machine learning and over-the-air (OTA) software updates as standalone applications, we interpret the core functionalities of these applications as advanced features. The essence of our approach lies in transforming the advanced functionalities of next-generation applications into features, laying the basis for feature-based next-generation embedded software. Through this, we aim to construct feature models that serve as blueprints for developing adaptable and scalable embedded software.

However, integrating new features into legacy applications presents another layer of challenges. Striking a balance between innovation and preservation demands an intricate understanding of the existing codebase. This integration requires careful planning for modularity, feasibility assessments, and incremental feature implementations. Such challenges highlight the need for structured frameworks to navigate this intricate landscape, ensuring compatibility and stability while promoting modernization.

5.2 FeaMod: A Framework for Enhancing Modularity, Adaptability and Code Reuse

This chapter introduces FeaMod, a framework conceived to address the outlined challenges. FeaMod integrates feature-based modularity with adaptive feature modeling to facilitate efficient embedded software design. At the core of FeaMod is the idea of understanding next-generation applications' functionalities as distinct features, which can then be modularized and integrated into diverse software environments. One of the steps of our approach is the transformation of non-modular code into feature-based modular code. This transformation based on advanced static code analysis, notably leveraging the BERT model. The precision of the BERT model enhances the accuracy of feature identification, ensuring contextually relevant and comprehensive feature extraction.

By aiding the identification and abstraction of computational features from existing codebases of next-generation applications, our methodology facilitates an environment conducive to software reuse and dynamic configuration. This, in turn, lays the foundation for designs that are not only flexible and maintainable but also testable [150], leading to reduced development time and costs.

Central to FeaMod is its unique adaptive feature model that encapsulates computational features, permitting dynamic configuration and system integration tailored to evolving requirements. Recognizing the dynamism required in the realm of embedded systems, we have introduced a set of rules governing feature relationships. These rules are integral to our approach, endowing the model with the flexibility to adapt to changing system requirements, user preferences, and environmental conditions.

This chapter's primary contributions encompass:

- The introduction of FeaMod, a holistic framework that synergizes feature-based modularity and adaptive feature modeling. This integration is made more robust with the precision of the BERT model.

- The presentation of a rule set that facilitates the identification of features, relationships, and constraints from source code, leading to the construction of the Adaptive Feature Model (AFM). This ensures adaptability and flexibility in response to shifting system requirements.
- A discussion on transforming next-generation applications, like machine learning and fog computing, into modular features. This perspective ensures that next-generation application functionalities are seamlessly integrated into embedded software environments, paving the way for efficient development, testing, and, consequently, more reliable systems.

The subsequent sections of this chapter will provide a deep dive into the FeaMod framework, exploring its methodologies, applications, and potential impact on the future of embedded software development.

5.3 FeaMod Framework: System Model and Assumptions

Our system model, illustrated in Figure 5.1, comprises various components that provide insights into the FeaMod framework. The FeaMod feature-based modeling framework distinguishes itself from traditional software development models by focusing on functionalities of next-generation applications, treating them as advanced features, and integrating the existing legacy next-generation application code. It employs automated feature extraction, modularity, and adaptability techniques to transform the code into a dynamic, easy-to-maintain format. Moreover, by reusing existing code, the design and development efforts are substantially reduced, resulting in significant efficiency gains. Transitioning from the foundational concept of modularity, a *feature* in embedded software is denoted as a distinct unit of functionality encapsulated as a computational unit or function [3] in any next-generation application.

Adaptive Feature Model (AFM): Moving beyond the static nature of traditional feature models, the AFM offers dynamic adaptability, allowing features to modify functionalities in real-time based on various contextual variables, thereby enhancing the system's responsiveness.

Despite numerous methods existing for locating and retrieving higher-level concepts from source code, most of them are manual or partially automated for building feature models directly from the source code. Therefore, the challenge lies in developing a framework that can facilitate automated feature extraction from existing software artifacts and construct an adaptive feature model, while also managing the translation of non-modular code into modular structures. This challenge can be formalized as follows:

Let P be a set of programs, F be a set of features in a program $p \in P$, and Ω be the set of all modularized

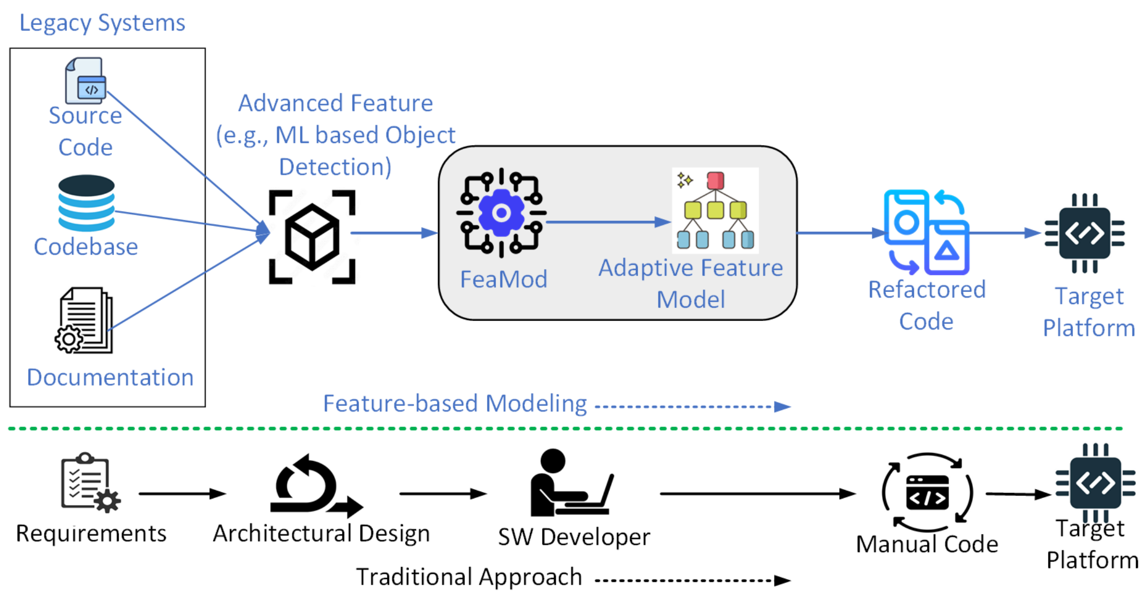


Figure 5.1: System model architecture (Feature-based vs Traditional software development approach)

versions of programs in P. The problem is to find a function $\phi : P \to \Omega$ that transforms each program into modular code, and a function $\psi : P \to 2^F$ that extracts the features from each program. Further, we also aim to create a function $\xi : 2^F \to 2^M$, where M is the adaptive feature model constructed from the features F. The ultimate challenge is to develop the FeaMod framework that enables ϕ, ψ, and ξ to automate the conversion of non-modular code into modular structures, extract features, and construct an adaptive feature model, respectively.

Assumptions

- We target non-modular Python programs used in advanced tasks such as machine learning-based object detection. Although C is prevalent in embedded systems [2], we opted for Python, given its growing relevance in next-generation applications.
- Utilizing the CodeBERT, a derivative of the BERT model fine-tuned for code analysis, leverages its strength in understanding code context and language tasks, thereby offering a robust solution for our feature extraction processes based on its proven proficiency in contextual embeddings derived from natural language understanding.
- The codebase can be parsed into an Abstract Syntax Tree (AST) using the Python `ast` or similar

module.

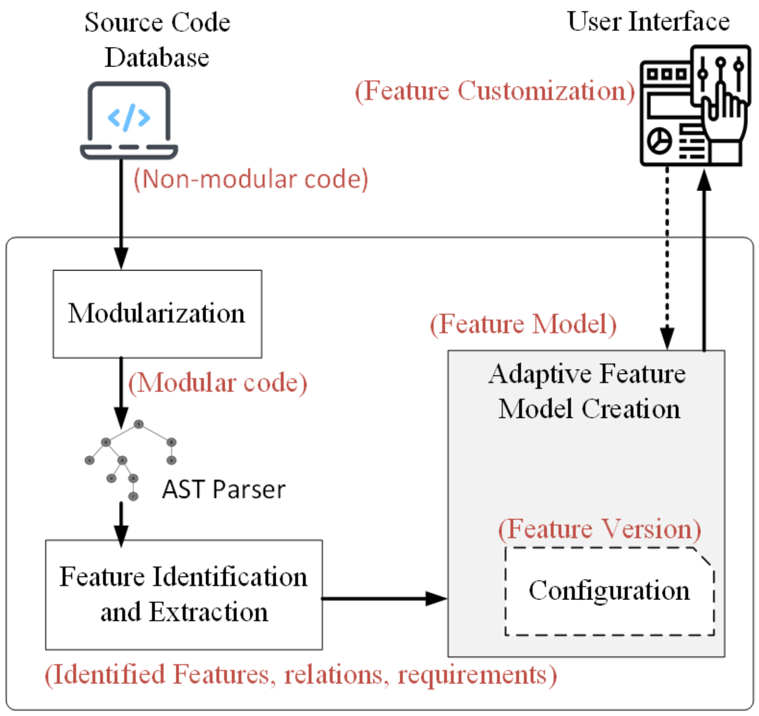


Figure 5.2: Proposed FeaMod framework

5.4 Detailed Methodology of the FeaMod Framework

The FeaMod framework utilizes modular and feature-oriented programming principles to identify features and requirements from source codes of advanced or next-generation applications. This adaptive methodology, comprising modularization, feature identification, and adaptive feature model creation, is delineated in the subsequent subsections and illustrated in Figure 5.2.

5.4.1 From Non-Modular Code to Modular Code: The Process of Modularization

In the initial phase of the FeaMod process, non-modular code is segmented into discrete modules using Python's AST module, which parses the source code into an abstract syntax tree. This tree aids in identifying and separating distinct tasks or related groups of tasks into individual functions, thus preserving the original logic while enhancing the code's readability and maintainability. The approach is designed to align with the following rules defined for transforming non-modular code into modular code.

Rule 1 - Reusability: Abstract similar code segments using metrics like cosine similarity on AST representations, facilitated by clone detection techniques.

Rule 2 - Cohesion: Group code performing a single task into one module, guided by variable usage frequency, function calls, and other dependencies metrics.

Rule 3 - Encapsulation: Bundle related data and functions into units using metrics such as Data Abstraction Coupling (DAC) that focus on the count of abstract data types in a unit.

Rule 4 - Conditional Segmentation: Modularize operations in conditional constructs using cyclomatic complexity to identify targets for modularization.

Rule 5 - Loop Abstraction: Abstract repetitive tasks in loops into separate functions, targeting complex structures identified through loop depth metrics.

Rule 6 - Loose Coupling: Enhance code independence by reducing dependencies between code parts, guided by Coupling Between Object (CBO) metrics.

Rule 7 - Error Handling: Standardize error handling by encapsulating common try-except blocks, using frequency metrics to streamline common patterns.

Rule 8 - Grouped Imports: Consolidate common imports into single modules, employing frequency metrics to create logically grouped modules with descriptive names.

Listing 5.1: Non-Modular Object Detection Code Snippet

```
# Start time for object detection
start_time = time.time()
model = cv2.dnn.readNetFromCaffe("deploy.prototxt",
    "res10_300x300_ssd_iter_140000.caffemodel")
image_path = "input.jpg"
image = cv2.imread(image_path)
blob = cv2.dnn.blobFromImage(image, 1.0, (300, 300), (104.0, 177.0, 123.0))
model.setInput(blob)
detections = model.forward()
for i in range(detections.shape[2]):
    confidence = detections[0, 0, i, 2]
    if confidence > 0.5:
        ...
mean_confidence = np.mean([detections[0, 0, i, 2] for i in range(detections.shape[2])])
try:
    print(f"Mean confidence for image {image_path}: {mean_confidence}")
except Exception as e:
    print(f"Error: {e}")
cv2.imshow("Output", image)
# End time for object detection
end_time = time.time()
print(f"Detection time: {end_time - start_time} seconds")
```

Listing 5.2: Refactored Modular Object Detection Code Snippet

```
def load_model(): # Rule 1
    return cv2.dnn.readNetFromCaffe("deploy.prototxt",
        "res10_300x300_ssd_iter_140000.caffemodel")
def load_image(image_path): # Rule 1
    return cv2.imread(image_path)
def process_image(image): # Rule 2
    return cv2.dnn.blobFromImage(image, 1.0, (300, 300), (104.0, 177.0, 123.0))
def detect_objects(model, blob): # Rule 2
    model.setInput(blob)
    return model.forward()
# Rule 4 and Rule 5
def display_detections(image, detections):
    for i in range(detections.shape[2]):
        ...
    cv2.imshow("Output", image)
```

```
def mean_confidence( detections ): # Rule 2
    return np.mean([ detections [0, 0, i , 2] for i in range( detections .shape [2]) ])
def display_mean_confidence (image_path, mean_confidence): # Rule 7
    try :
        ...
def main(image_path): # Rule 6: Loose Coupling
    model = load_model()
    image = load_image(image_path)
    blob = process_image(image)
    detections = detect_objects (model, blob)
    ...
main("input .jpg")
```

In transforming a non-modular code segment (e.g., Listing 5.1) into a modular like `load_model(),` `load_image(image_path)` shown in Listing 5.2, we start by applying transformation *Rules* to pinpoint different parts of the code. Utilizing Python's `ast` module, we parse the code into an Abstract Syntax Tree (AST). Following this, we identify and extract the nodes corresponding to the reusable code segment, which forms the body of a new `ast.FunctionDef` node. This node represents our new function. This new function encapsulates the reusable operations, accepts arguments, and returns the outcome of the encapsulated code. Subsequently, this function is inserted into the AST in place of the original code segment. The final step involves unparsing the modified AST back into source code using the `astunparse` module. As a result, we obtain the `load_model() and load_image(image_path)` function that encapsulates the original code and can now be invoked wherever required, thus enhancing the code's reusability and maintainability.

The code snippet in Listing 5.2 presents several functions or features, each dedicated to a specific operation, such as loading a machine learning model (`load_model`), reading an image file (`load_image`), processing the image (`process_image`), identifying objects in the processed image (`detect_objects`), and visualizing the detection results (`display_detections`).

5.4.2 Techniques for Feature Identification and Requirements Extraction

Identifying Functional Features

Every function in the codebase denotes a distinct feature, representing a unique operation or functionality within the system. Leveraging the BERT model fine-tuned with domain-specific semantics aids in extracting potential features from function names and global variables, creating a semantic space to identify functional and non-functional requirements. We apply established classifications to characterize these features systematically [3], i.e., *Mandatory*, *Optional*, *OR*, *XOR*, *Requires*, and *Excludes*. The FeaMod framework uses a set of rules to categorize features into:

- Mandatory: A feature that must be included whenever its parent feature is selected (e.g., `load_model`).
- Optional: May or may not be present depending on the conditions (e.g., `display_mean_confidence`).
- OR: One or more among the child features must be selected if the parent is included.
- XOR (Alternative): Exactly one among the child features can be chosen when the parent is selected.
- Requires: The presence of one feature necessitates the presence of another.
- Excludes: The presence of one feature prohibits the inclusion of another.

Identifying Non-Functional Features

To identify non-functional features in software artifacts, we employ a fine-tuned BERT model, taking advantage of its deep understanding of context and semantics, as detailed in Algorithm 10. The process starts with the tokenization of each document d_i in set D, followed by forming a dictionary $Dict_{d_i}$ to map each token to its frequency in d_i. We then use TF-IDF scores to select the top N tokens as candidate features (CF_{d_i}) [18].

Next, the BERT model creates embeddings E_{t_j} for each token in CF_{d_i}, which are clustered to group semantically similar tokens. A representative token RT_{C_k} from each cluster is chosen as a potential non-functional feature.

Example: For instance, in a source code with frequent mentions of the term `security`, our method initially recognizes it as a candidate feature through high TF-IDF scoring. BERT then discerns different contexts of the term, like `data security` and `network security`, and groups them under the broader feature - `security`.

Algorithm 10: Identifying Non-Functional Features

```
Input: $D = \{d_1, d_2, ..., d_n\}$, a set of documents including code artifact, associated comments
Output: Potential non-functional features
1  for each $d_i$ in $D$ do
2      Tokenize $d_i$ into separate identifiers (tokens);
3      Create a dictionary $Dict_{d_i}$ mapping each token;
4      Calculate TF-IDF score for each token in $Dict_{d_i}$;
5      Select the top $N$ tokens based on the TF-IDF scores as candidate features; set is $CF_{d_i}$;
6  end
7  Fine-tune BERT using domain-specific data (if available);
8  for each $d_i$ in $D$ do
9      for each token $t_j$ in $CF_{d_i}$ do
10         Create an embedding $E_{t_j}$ using BERT;
11     end
12     Cluster the embeddings $\{E_{t_1}, E_{t_2}, ..., E_{t_n}\}$ to group semantically similar tokens, obtaining
         clusters $C_1, C_2, ..., C_m$;
13     for each cluster $C_k$ do
14         Identify the representative token $RT_{C_k}$ from $C_k$ as a potential non-functional feature;
15     end
16 end
17 return Non-functional features for each document;
```

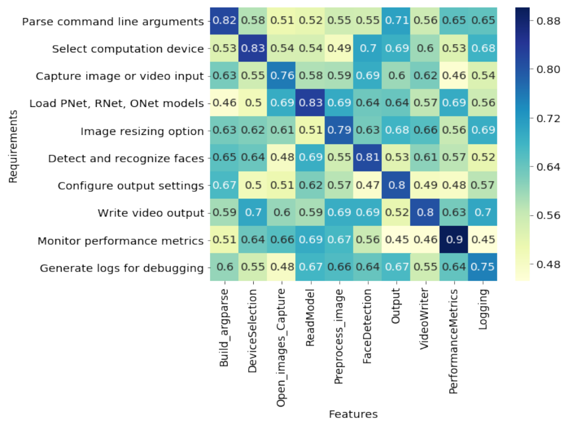


Figure 5.3: Features and requirements map for face detection

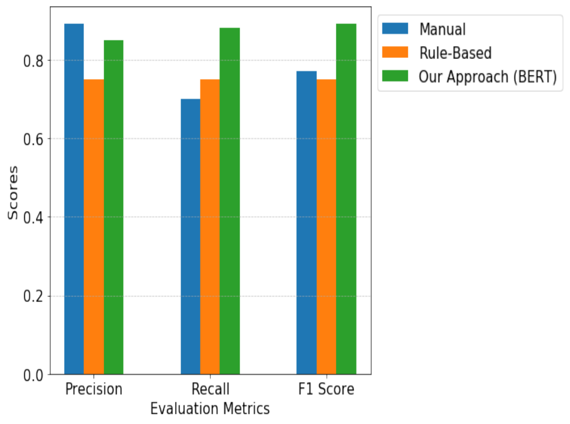


Figure 5.4: Comparison of BERT with Rule-based approach

BERT-enhanced Requirements Extraction

Leveraging BERT's semantic analysis capabilities, we delineate a strategy for extracting both functional and non-functional requirements efficiently from software artifacts.

- Functional Requirements Extraction: These encapsulate the core functionalities that the system is designed to execute.
  - Extracting Conditional Constructs: We identify functional requirements by analyzing conditions in `if`, `while`, and `for` loops using BERT, facilitating a deep semantic understanding.
  - Function and Method Analysis: We utilize BERT to interpret function names and method signatures, helping to identify potential functionalities even in ambiguous scenarios.
- Non-functional Requirements Extraction: These involve system attributes such as performance and security.
  - Extracting Comments: BERT assists in analyzing comments semantically to pinpoint non-functional requirements effectively.
  - Documentation Analysis: Non-functional aspects are often detailed in READMEs and architectural documents.
  - Code Patterns and Libraries: Specific code patterns and libraries can signal non-functional requirements, such as security considerations.

5.4.3 Building an Adaptive Feature Model

The Adaptive Feature Model (AFM) creation facilitates dynamic reconfiguration of feature models, offering enhanced flexibility and efficiency by adapting to variations in system state and other factors. Considering a system with features denoted as f_i, we introduce the Feature Interaction Graph (FIG) $G = (V, E)$ to represent feature interactions, where where the vertices $V = f_1, f_2, \ldots, f_n$ represent features, and edges $E = e_{ij}$ represent interactions between features. For each feature f_i, we define an interaction set I_i, which includes all the features that interact with f_i. The construction of the AFM involves:

1. **Construction of the FIG:** Identifying feature interactions through code analysis and representing them in the FIG.
2. **Weight Assignment:** Quantifying feature interactions using metrics such as call frequency or parameter interactions and assigning weights to the edges in the FIG.
3. **Feature Clustering:** Utilizing graph clustering algorithms like K-means to identify groups of closely related features, which form subsystems in the final model.

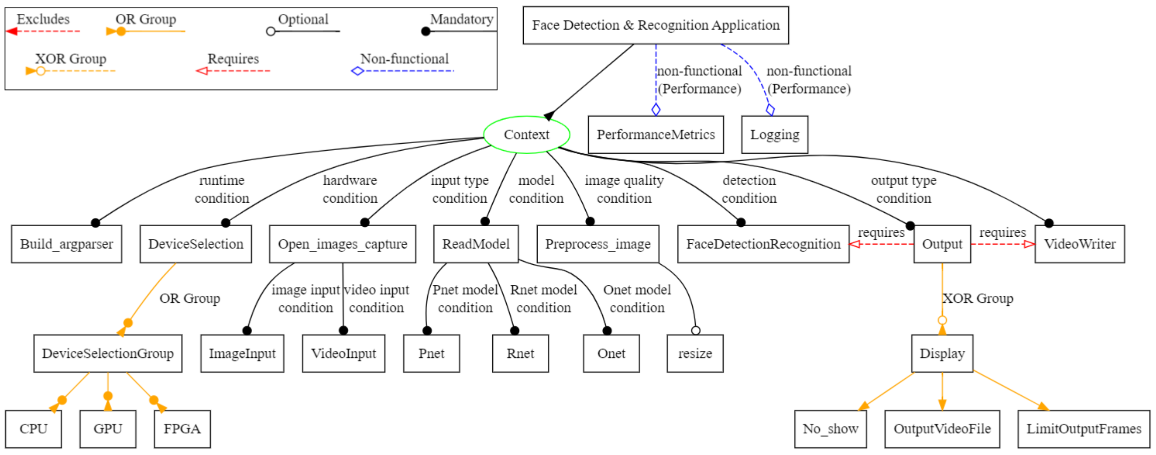


Figure 5.5: Adaptive feature model for face detection

4. **Feature Configurations:** Each feature f_i has an associated set of possible versions, denoted as $FC_i = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$. Given the dynamic nature of software systems and ever-evolving requirements, feature versioning is a critical component in the feature model, enabling it to manage complex, changing requirements in distributed systems efficiently. A feature model configuration is considered valid if it includes all the feature configurations on which it depends.

Table 5.1: Feature model statistics for face detection

Feature Model Attribute	**Case Study [151]**
Number of functional Features	22
Number of Non-Functional Features	2
Number of Constraints	2
Number of Valid Feature Configurations	42

5.5 Evaluation of the FeaMod Framework

To evaluate the FeaMod framework, we utilized the publicly accessible Face Detection GitHub repository [151] as a case study. The analysis focused on the framework's proficiency in identifying features, recognizing feature relationships, and setting up configurations, coupled with assessing the adaptive feature model's construction time.

Case Study: Feature-Based Modularization in Face Detection

The FeaMod, powered by the Graphviz library for visualization, employed a series of steps to generate the application's Feature Model. Initially, the code was modularized to isolate specific features and make their functions more distinct and manageable. Subsequently, FeaMod identified the key features of the application, such as *DeviceSelection*, *Open_image_capture*, *Read_Model*, *Preprocess_image*, *FaceDetectionRecognition*, *Output*, and *Video Writer*. These features were categorized as mandatory, optional, or non-functional based on their roles within the application. Two non-functional features, *Performance Metrics* and *Logging*, were identified using the TF-IDF and BERT as discussed in Algorithm 10. *Performance Metrics* and *Logging* are classified as non-functional features because they address the system's operational qualities like efficiency and maintainability rather than its core functionalities. For example, *Logging* library provides a mechanism for tracking the system's behavior, errors, and operational data, facilitating debugging and system monitoring, which improves maintainability and reliability but is not part of the system's primary functional processes. To validate the effectiveness of our approach in identifying the features and associated requirements, a cosine similarity analysis was performed, the results of which are depicted in Figure 5.3. The diagonal elements of the cosine similarity matrix, representing the similarity scores between each feature and its corresponding requirement, were notably higher. This indicates a strong correlation between the identified features and their corresponding requirements, highlighting the accuracy of the proposed method in capturing the semantic relationship between features and requirements.

In this face detection case study, the manual approach identified 26 features and 39 requirements. Our BERT-enabled approach successfully identified 22 of these features and 34 requirements, demonstrating a balanced performance with high precision and recall, as detailed in Figure 5.4. This approach outperforms the rule-based method, achieving F1 scores of 86.5% and 86% for feature and requirement identification, respectively, and showcasing its effectiveness in deriving insightful results through the automated analysis of source code and comments. This underlines the potential of our approach in facilitating more accurate and comprehensive feature and requirement identification in software development.

The relationship and constraints among the features were then established. For instance, *Output* was found to require *FaceDetectionRecognition* and *Video Writer* for it to function correctly. The *DeviceSelection* feature was determined to have an *OR* relationship among its sub-features: *CPU*, *GPU*, *FPGA*. The constructed adaptive feature model is shown in Figure 5.5. To ensure feature configuration, FeaMod checks for the existence of required features and the consistency of constraints among features, such as *OR*, *XOR*,

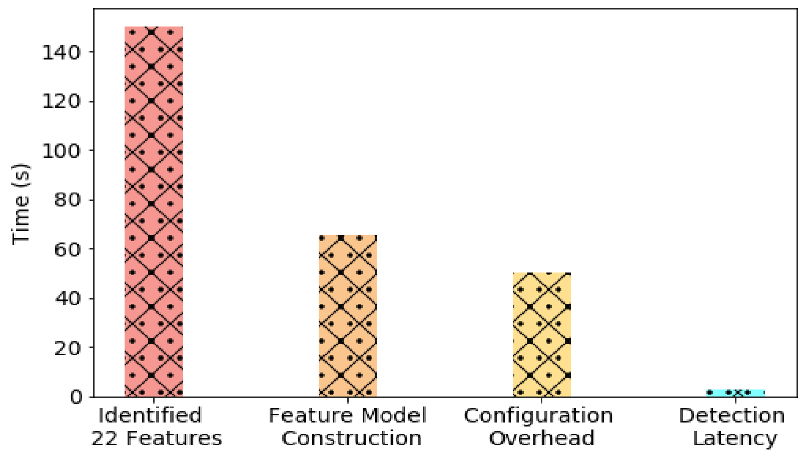


Figure 5.6: Execution time compare for different attributes using FeaMod

and exclusionary relationships.

Table 5.1 delineates the breakdown of different feature model attributes of the face detection application examined in case study 1. The execution times for the various attributes of the FeaMod are illustrated in Figure 5.6. The most time-intensive task was the identification and extraction of features, taking approximately 150s — a reflection of the computational demands of discerning 22 features. Following this, the construction of the feature model necessitated 65s, and ensuring feature model configuration overhead consumed around 50s. Detection latency remained minimal at a mere 2.5s. Despite the computational demands, the data manifests the potential for further optimization in performance, specifically in the realms of feature identification and model construction.

5.6 Summary

In the field of adaptive feature modeling, our framework sets itself apart with its unique utilization of the BERT model to provide a dynamic and adaptable feature modeling process. While other researchers like Alam *et al.* [52] tackled concern-driven strategies, FeaMod stands out with its component functionality-centered perspective. The landscape of adaptive feature modeling is filled with various innovative approaches, especially in its capacity to adjust to changing system requirements.

The proposed FeaMod framework, a feature-based modularity and adaptive feature modeling synthesis, emerges as a transformative solution in embedded software development. To reuse advanced features code, our framework shows how to identify functional and non-functional features from source codes and their interdependencies. This offers a fresh perspective on code reuse and dynamic adaptability. With the potential to reshape the embedded software development landscape, our framework holds promise for future refinements and broader applications. As we move forward, it will be exciting to see how this framework evolves and what new horizons it might explore.

By bridging gaps left by previous works and introducing new methodologies, FeaMod stands as a testament to the continuous evolution of software development practices. The journey ahead holds immense promise, with opportunities for further refinement and expansion of the framework's applications.

Chapter 6

RESULTS ANALYSIS AND DISCUSSION

6.1 Introduction

This chapter is dedicated to addressing and clarifying the research questions that are mentioned in the Introduction chapter 1 of this thesis. These questions relate to code reuse and advanced application requirements in designing and developing next-generation embedded software, particularly in understanding features and requirements. Alongside the evaluation experiments presented in chapters 3 to 5, we conduct additional experiments to clarify these research questions further. We undertake a detailed analysis of these additional experiments, which are conducted across two primary categories:

a) Investigating legacy embedded software to facilitate the design and development of next-generation embedded systems.
    i) Feature model creation from legacy software
    ii) Analysis of feature search time within the feature model

b) Integration of fog computing architecture for next-generation embedded applications.
    i) Design and implementation of a testbed for the proposed OTA update algorithm.
    ii) Optimal partitioning of ML models for enhanced performance.

6.2 Investigate Legacy Software for Design and Development Embedded Software

Feature Model Creation from Legacy Software

To apply our proposed approach, we select a GitHub project "Software Controller for Vehicles" [152] written in C programming language. This GitHub project is the demo implementation of a car's software systems, and we use it to demonstrate our proposed method. To obtain each function's information, we run the function flow graph generator *cflow* and generate a function list. The C keywords and library functions (e.g., malloc, fopen, printf) are ignored to simplify the process.

From the function call graph, we extract 38 reusable functions as features for the implementation of a car's software systems. According to our proposed approach, initially, we find the mandatory and optional features where we retrieve 21 *mandatory* features and eight *optional* features. For example, feature `car_software` has one *mandatory* feature `system` and one *optional* feature `test`. After that, we identify the feature relationships defined in Chapter 2 and we get four *alternatives*, and four *OR* relations feature relationships. The maximum level of each feature is extracted from the call graph and determines the potential parent and child list for each feature. After that, we apply the feature model construction actions discussed in Chapter 3 to update the placement of features in different levels if required.

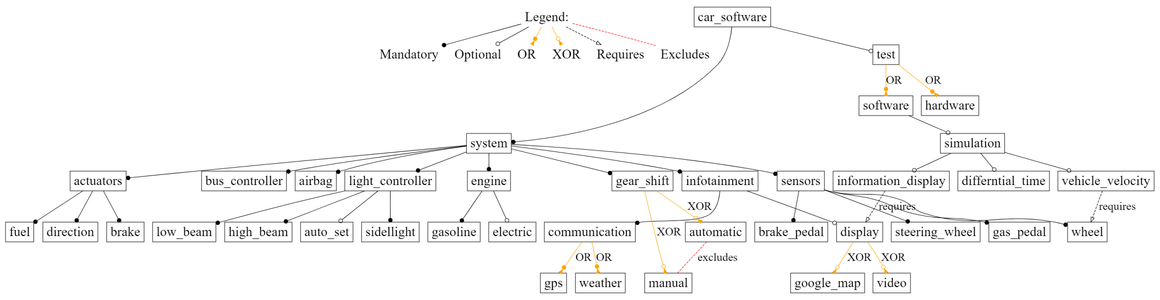
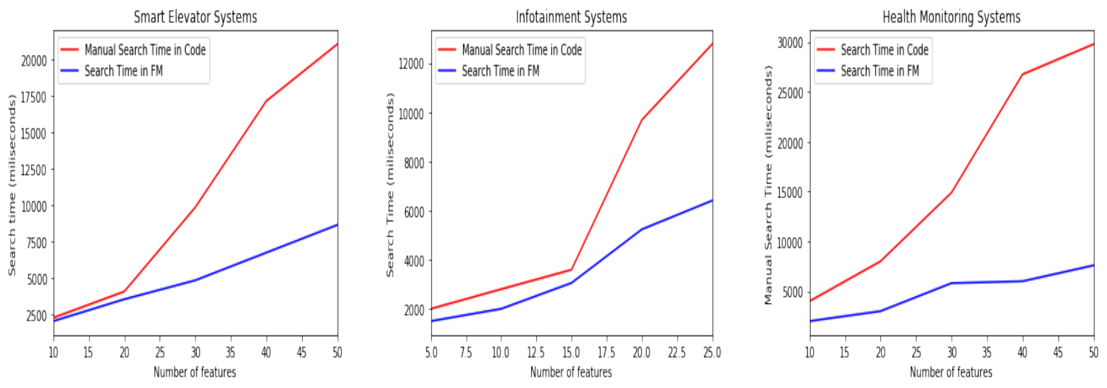


Figure 6.1: Feature model creation from GitHub project "Software Controller for Vehicles" [152]

The output of our proposed approach for "Software Controller for Vehicles" is shown in Figure 6.1. In this figure, each feature is associated with its parent and child nodes. The hierarchy of the parent-child features is shown based on the level of each feature, where `car_software` is placed at Level 0 as a root node. Besides, we find the *requires* and *excludes* relationships to understand the dependency of the features. To test the feature called `vehicle_velocity`, it needs *requires* relation with another feature `wheel` that counts the wheels' rotation. Similarly, *manual* and *automatic* features are mutually exclusive. One feature excludes other features.

A feature may have or may not have constraints. We use the C parser *pycparser* to generate the Abstract Syntax Tree (AST) and extract the code artifact along with constants, variables, as well as constraints. As an example, `infotainment` systems and `light_controller` are features without constraints. These features may be called periodically/aperiodically with user inputs. On the other hand, features with constraints are `engine`, `gear_shift`, `airbag`, `brake`, etc. These features require validating certain constraints to ensure the safety of the system. The necessary test cases need to be executed before adding any feature in the model to satisfy the constraints.

Feature Search Time Analysis



(a) Feature search in smart elevator system

(b) Feature search in infotainment system

(c) Feature search in health monitoring system

Figure 6.2: Search time comparison for finding a feature (deepest leaf node) among different GitHub embedded systems projects

In the experiment, we collect three different GitHub projects which a) smart elevator system, b) infotainment system, and c) health monitoring system. We create an individual feature model for each application and calculate the search time to find a feature located in the deepest leaf node. Figure 6.2 shows the comparison of search time among these three different GitHub embedded systems projects. In all three projects, we observe that the search time for finding a selected feature using the feature model is much lower than the manual search in the codebase. The manual search reads the codebase sequentially, starting from the first line to the end. In Figure 6.2, the x-axis represents the number of features and the y-axis represents the required search time for finding a feature over a varying number of features.

Moreover, the experiment result states that the searching times vary for changing the number of features. Although the search time for a small embedded system application does not vary much, our approach outperforms the manual search approach for a large-scale project with a large number of features. In our approach, the feature search times for three different projects (smart elevator system [153], infotainment system [154], and health monitoring system [155]) get reduced by 59.0%, 49.8%, and 74.5% for 50, 25, and 50 number of features, respectively.

6.3 Integrating Fog Architecture for Next-generation Embedded Application

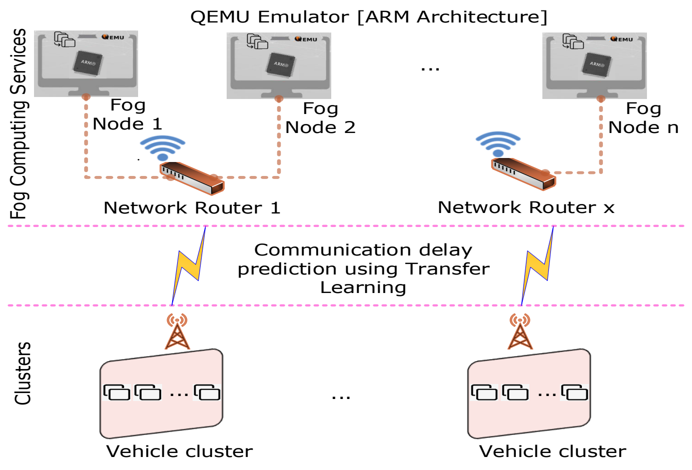


Figure 6.3: Our experimental testbed design for faster OTA updates

6.3.1 Design and Implementation of a Testbed for Fog-enabled OTA Update Algorithm

In order to show the efficacy of our proposed OTA update algorithm in the real-world, we design and implement a testbed. The testbed architecture is shown in Figure 6.3. The architecture incorporates fog nodes that provide OTA software updates to various vehicles. The network communication delay between the fog nodes and the vehicles is calculated using the proposed transfer learning method. The fog nodes are connected through network routers that operate at different WiFi ranges. Fog nodes and vehicles have been implemented as Virtual Machines (VMs), using the QEMU virtualizer. These VMs are executed on different laptops. The QEMU emulates the ARM hardware architecture, which has gained popularity in the automotive industry for its reduced production cost and power efficiency. For example, the R-Car H3 provides computational services to in-vehicle infotainment and safety support systems [156]. It uses the Arm Cortex-A53 CPU architecture which implements the widely deployed ARMv8-A 64-bit instruction set.

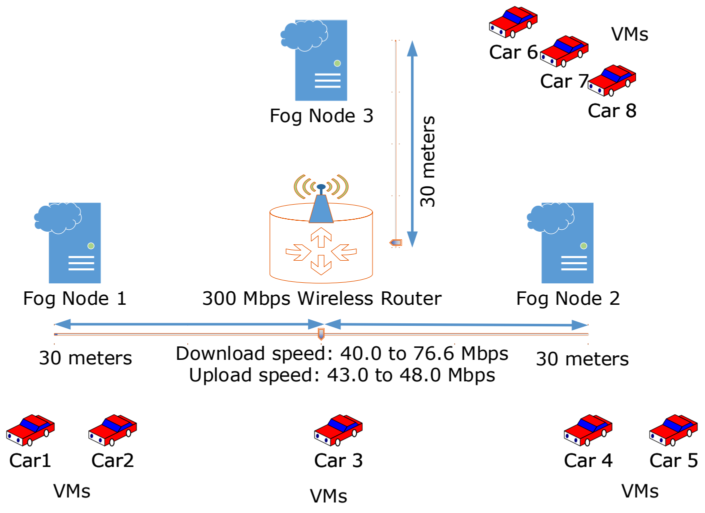
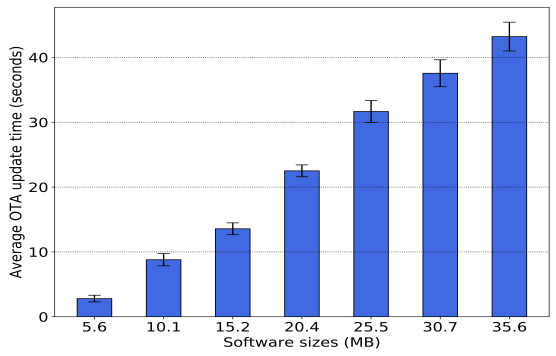
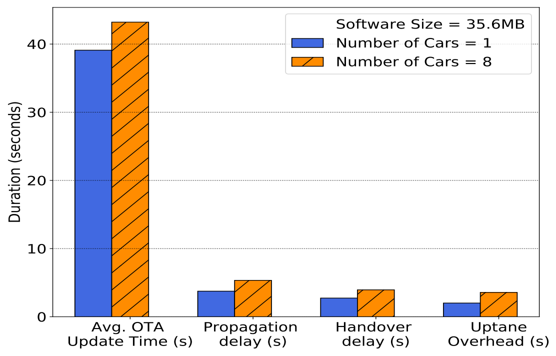


Figure 6.4: OTA updates for three fog nodes and eight cars in our experimental testbed

We implement our experimental testbed which includes software, hardware, and networking components. We integrate a pre-existing OTA update framework called *Uptane* [89] into our testbed. Figure 6.4 shows the experimental setup of our proposed architecture. We create multiple VMs and assign them as different fog nodes and vehicles/cars. Each VM is configured with 4GB RAM and runs Linux. The connection between a vehicle and the fog node is created through a 300Mbps N TPLink wireless router (TL-WR841N), which uses Transmission Control Protocol/Internet Protocol (TCP/IP) for data transmission. The maximum coverage area for each router is roughly 70 meters. We create three fog nodes (1, 2, and 3) and assign them at a distance of 30 meters from the router, as shown in Figure 6.4. We assume that these fog nodes are uniformly distributed in a cluster. A total of eight (8) VMs are created and considered as vehicles in a cluster. Though we consider cars, the work can be applied to any vehicle that requires OTA software updates: trucks, ships. We assume that the cars are moved at a speed of 1.35m/s, for 30 meters. While passing the maximum coverage area of a fog node, a car performs a handover to connect with the nearest available fog nodes. We adjust our algorithm to select a fog node that is near to the vehicle and has a strong network connection. The download speed for data transmission is between 40Mbps to 76.6Mbps, and the upload speed is between 43Mbps to 48Mbps.



(a) OTA update time (T_i) calculation for various sizes of software (S_i^{size}) at 95% confidence interval



(b) Comparison of OTA update time (T_i), handover delay (Δ_i), propagation delay(T_i^p) and overhead for using *Uptane*

Figure 6.5: OTA update time and delay comparison for various software and traffic sizes

Afterwards, we perform an experiment to update software of various sizes using Algorithm 7. We consider software sizes between 5MB to 40MB and calculate the average OTA update time. The experimental results are shown in Figure 6.5(a). We observe that the required OTA update time is much shorter for a smaller size of software and vice versa. For a small software size of 5.6MB, the average OTA update time is around 2.80s with a 95% confidence interval and $\pm$0.51s margin of error. On the other hand, the average OTA update time for a large software size of 35.6MB is around 43.21s with a 95% confidence interval and $\pm$2.22s margin of error. We observe that the margin of error for a large software size is higher than that of a small software size. Propagation and handover delay are the main causes of high margin of error for larger software updates. An update of a smaller size software less likely causes a handover situation, and therefore the error remains low. Moreover, larger software requires more transmission time, where propagation delay and handover delay increase proportionally with size. Apart from the calculation of OTA update or software download time, we also measure the overhead of using the *Uptane* framework.

In order to observe the effect of number of vehicles on OTA updates, we calculate the total OTA update time with the software of size 35.6MB for one car and eight cars. We calculate the propagation delay after 10 repeated runs in updating a fixed size of the software. We take the maximum propagation delay from these repeated runs and define the value as the worst-case propagation delay (see Algorithm 6). According to our experiments, the propagation delay is about 3.75s when there is only one car carrying out the software update and 5.34s when eight cars perform the updates at the same time. Figure 6.5(b) presents the comparison among OTA update time, propagation delay, handover delay and the overhead for using the *Uptane* framework.

Results show that the propagation and handover delays are higher for eight cars compared to one car. This is expected because we notice similar outcomes in our simulations. Our small-scale experimental analysis on the testbed partially validates the simulation results we obtain and thus demonstrates that our proposed mechanisms offer faster fog computing based OTA updates using transfer learning.

6.3.2 Optimal ML Model Partitioning for Accelerated Performance

Experimental Setup: The experiments have been conducted on the Linux 5.1.0-52-generic (x86_64) kernel with Ubuntu 20.04 LTS operating system. We consider that our embedded system has general-purposed processors for running DNN applications. The CPU is configured with Core i5-9400F, a 2.90GHz processor, 16GB RAM, and six cores with 6x4100.00MHz. However, we also evaluate the performance of our proposed framework for different architectures (e.g., AlexNet [44], ResNet [133], VGG-16 [134]) in a NVIDIA Jetson Nano [135] Quad-core ARM Cortex-A57 processor with the same operating systems. The Jetson Nano is equipped with 4 GB of 64-bit LPDDR4 memory. The proposed approach has been implemented in python3 using TensorFlow-Keras, whereas AlexNet has been used as the underlying DNN architecture.

ImageNet Dataset: In this implementation, the AlexNet Convolutional Neural Network (CNN) architecture is utilized for image object classification. This architecture is trained on the ImageNet dataset [157], which originally consists of millions of training images classified into 1000 classes. However, in this experiment, a customized dataset is created using 2000 training images and 1000 validation images, with a reduced number of target classes limited to two for running the experiment with different scenarios. The model comprises five convolution layers, five max-pooling layers, and three fully-connected layers, and is trained for 100 iterations (epochs) to evaluate its performance.

Finding Optimal Splits

The performance of a deep neural network (DNN) model's training is closely tied to the partitioning strategy during parallel computation. In our study, we use the AlexNet CNN architecture and partition it into various sub-modules based on layer dependencies and the number of available computing devices. We conduct experiments with four scenarios, splitting the network into zero, two, three, and four parts. Using the available six processing cores, we experimentally analyze these optimal sets returned by Algorithm 8.

We first run the complete AlexNet model on a single CPU core as a baseline without splitting the network. As shown in Table 6.1, the net execution time was 7.65 x 10^3 seconds for training the model over

Table 6.1: Comparing DNN model (AlexNet) parallel training for different numbers of splits (epochs = 100, batch size = 128)

Network Execution	**Number of Splits**	**Split Positions (Conv-FC)**	**Net Execution Time (sec)**	**CPU Cores**	**Memory Usage (%)**
Sequential Model	0	0	$7.65x10^3$	1	88%
Data Parallelism	2	Conv1 to 5 FC 1 to 3 (x neurons)	$4.60x10^3$	2	47%
Pipeline Model Parallelism	3	Conv1 to 5 FC 1 to 3 (x/2 neurons) FC 1 to 3 (x/2 neurons)	$4.21x10^3$	3	60%
Pipeline Model Parallelism	4	Conv1 to 3 Conv 4 to 5 FC 1 to 3 (x/2 neurons) FC 1 to 3 (x/2 neurons)	$4.33x10^3$	4	32%

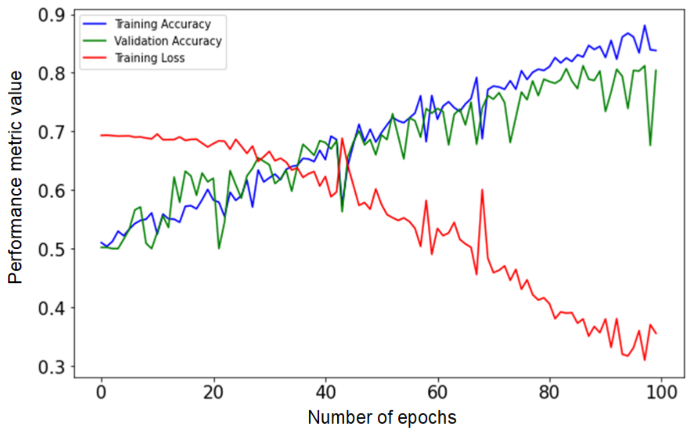
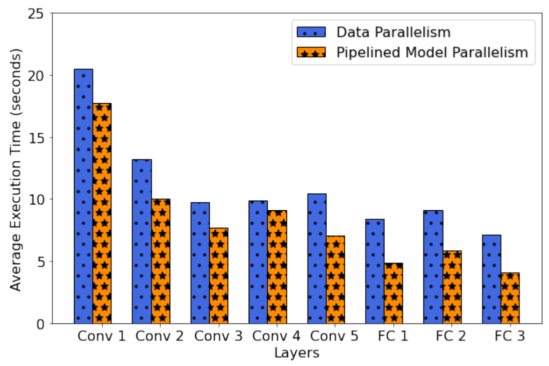


Figure 6.6: Training and validation accuracy for AlexNet pipeline model parallelization

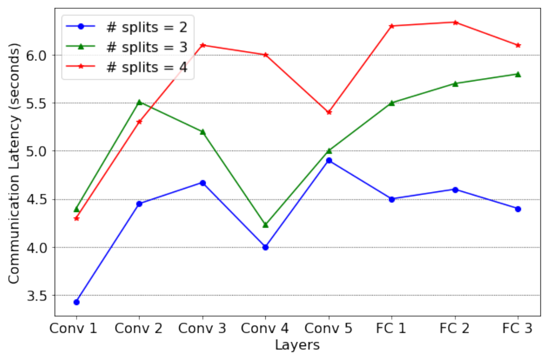
100 epochs with 128 mini-batch sizes. In the next scenario, we created two networks that replicated the AlexNet model, with each network containing both the convolution and fully connected layers. To take advantage of data parallelism, we executed the convolution layers in a pipelined manner (as shown in Figure 4.17) and executed the fully connected layers sequentially. This approach reduced the computation time by nearly 3.05 x 10^3 seconds and reduced memory utilization to under 50%, highlighting the possibility of poor resource utilization.

Next, we apply our proposed Algorithms 8 and 8 to determine the optimal number of splits for six CPU cores. The results indicate that the possible splits are three and four, as they result in a lower net execution time compared to other numbers of splits. As shown in Table 6.1, the net execution time of the pipeline model parallelism is lower for three and four splits compared to the other configurations. However, it is worth noting that the memory usage for three splits is approximately 12% higher than for four splits. Higher memory usage is often preferred for model partitioning as it can improve performance, load balancing, scalability, and avoid underutilization of resources. This suggests that the optimal number of splits for AlexNet with this experimental configuration should be three. Hence, this approach implements data parallelism in multi-core processors for the convolution layers and model parallelism for the fully connected layers.

To further evaluate the performance of the optimal number of splits, i.e., three, we conduct a training experiment using the proposed pipeline model parallelism approach on the ImageNet dataset with AlexNet as the CNN architecture. The training losses and optimization method, using stochastic gradient descent (SGD), are incorporated through the TensorFlow module. The comparison of training and validation accuracy is presented in Figure 6.6, with results showing training accuracy of around 88% and validation accuracy of approximately 80% after 100 epochs. In a separate experiment, our findings indicate that the AlexNet model without model parallelism achieved a training accuracy of approximately 90% after 100 epochs. Thus, our approach effectively reduces the computation time for model training without compromising accuracy.



(a) Average execution time (per epoch) comparison of each layer for AlexNet CNN architecture



(b) Communication latency comparison for splitting the AlexNet using pipeline model parallelism

Figure 6.7: Average execution time and communication latency comparison for splitting AlexNet

Layer Execution Time Comparison

In order to evaluate the effectiveness of the proposed pipeline model parallelism framework, we assess the average execution time of each layer of the AlexNet architecture in this experiment. One of the objectives of this paper is to minimize the net execution time by splitting the network into an optimal number of splits. Figure 6.7(a) illustrates the average execution times of different layers, calculated after 100 iterations. The workload is distributed with a batch size of 128. The convolutional layers require a longer processing time due to the high volume of parameters (e.g., 62,378,344) and weights they contain compared to the fully connected layers. As a result, the average execution times of the convolutional layers are comparatively higher than those of the fully connected layers. Furthermore, we compare the performance of the data parallelism with our proposed approach. The results show that the pipeline model parallelism reduces the average execution time when the model is split into three parts, with each layer having a lower execution time.

Communication Latency Comparison

In this experiment, we evaluate the impact of the number of splits on communication latency in parallel processing. To understand the communication latency between the layers, we measure the time required to transfer the first data packet from one layer to another during a single iteration. It measures the communication delay between two adjacent layers in a parallel processing system. We compare the communication latency for different numbers of model splits on the available CPU cores with varying frequency distributions. Figure 6.7(b) shows the comparison of latency among all the layers for various numbers of splits. The communication latency for two and three splits does not differ significantly, but a substantial increase in latency is observed when the number of splits increases to four. The latency for four splits was found to be over six seconds, as the number of computing processors increases with the number of splits. This experiment highlights the importance of finding the optimal number of splits to minimize communication latency in DNN parallel processing.

6.4 Threats to Validity

To understand the scope of our work, we report threats to the validity of our proposed approach.

6.4.1 Embedded Software Reuse

- *Scope and Generalizability:* While our methodology is grounded in the context of C code, its foundational principles bear the potential for adaptation to other object oriented programming languages. This adaptability is especially promising given the similarities in procedural or functional structures across various languages.

- *Identifying Non-Functional Features from Poorly Documented Code:* Identifying non-functional requirements in embedded systems arises from the quality of software documentation. We rely on developer comments in the code and libraries related to non-functional features as primary sources of information. However, suppose the program code is not adequately documented or fails to maintain the coding standards, especially lacking informative developer comments. There is a chance of overlooking necessary non-functional features in that case. This limitation could lead to an incomplete understanding of these requirements. We plan to incorporate dynamic analysis in our future work to overcome this. This would allow us to explore non-functional features more deeply, potentially uncovering aspects not immediately evident through static code analysis alone.

- *Model Biases and Refinements:* The inherent biases of language models and machine learning models, such as LDA and BERT, present challenges in the accurate interpretation of code artifacts. However, the avenue of fine-tuning them with domain-specific datasets offers a promising countermeasure, allowing these models to be more attuned to the nuances and specificities of the source code.

- *Reproducibility of the Experimental Results:* Given the unique nature of embedded software and the specific configurations of their development environments, replicating our findings in different settings might yield a different result. However, using uniformity in the versions of language models and the parameter settings of our machine learning models, we expect to achieve consistent results across similar experimental setups.

- *Detail Granularity:* The level of granularity in the feature extraction process emerged as a pivotal consideration. The flexibility to tailor extraction, be it from a broad overview or a granulated perspective, ensures that the process remains aligned with the desired level of intricacy.

The integration of advanced NLP models with static code analysis signifies a transformative shift in software engineering methodologies. By harnessing the semantic prowess of BERT in conjunction with topic modeling, we have charted new territories in requirements extraction and feature modeling. While the results are promising, they underscore the importance of a unified blend of traditional and contemporary techniques.

6.4.2 ML-based Fog Assisted Embedded Software Development

- *Can we generalize the approach for all DNN applications?:* In this work, we have worked especially on the CNN architecture, which is a particular type of DNN. Therefore, our model-splitting approach only applies to those architectures with a structure that is similar to CNN.
- *Is this approach applicable for GPU-enabled embedded systems?:* This work mainly focuses on supporting multiprocessor embedded systems that largely have multi-core CPUs. Although GPU support is open, the embedded systems community still needs multiprocessor CPU-enabled solutions to reuse the existing embedded systems, avoiding additional costs. However, this work is easily extensible to support multi-GPU embedded systems.
- *Does this approach tune the hyper-parameters during model training?:* The proposed approach only identifies how the model can be split onto multiple processors. Therefore, during our experiment, we use the default configurations of the hyper-parameters of all the CNN architectures. Thus, the result may vary if one tunes the parameters in different computing platforms. However, we can analyze this in the future to determine what hyper-parameters impact performance acceleration.

6.5 Clarifying Research Questions

In this section, we clarify how each research question mentioned in Section 1.3 has been tackled through various experiments and analyses conducted during this research. The findings are discussed in relation to each research question, offering clear insights into how the objectives of the thesis were met.

- **RQ1: Essential Criteria for Identifying Reusable Features from Legacy Embedded Software**

  The identification and extraction of reusable features from legacy embedded software were achieved using a combination of BERT and topic modeling techniques discussed in Chapter 3. In addressing RQ1, our methodology distinguished between functional and non-functional features within the legacy embedded software. For functional features, we adopted the principle that every user-defined function in the codebase represents a potential feature, each denoting a unique operation or functionality within the system. We prioritize user-defined functions in our analysis as they typically contain the core logic and unique features specific to the embedded software, providing a more accurate representation of its functionality.

However, to minimize the granularity of considering functions as features, we apply LDA and BERT on the Longest Common Unit (LCU) technique of function calls to find the potential features. The frequency of function calls and the presence of conditional statements within the code were analyzed, serving as key indicators of feature reusability and interrelationships. On the other hand, for non-functional features, we focused on program libraries and comments associated with each function. These elements often provide critical insights into the context, purpose, and constraints of the functions, thereby enriching our understanding of the software's broader architecture and design considerations.

Our examination of the "Software Controller for Vehicles" GitHub project (see Figure 6.1) demonstrated our ability to extract and identify 38 reusable functions. Utilizing tools such as *cflow* and *pycparser*, along with NLP models like BERT and topic modeling, we determine mandatory and optional features and their interrelations (OR, XOR, Requires, Excludes). This process validated our approach to identifying reusable features from legacy embedded software.

- **RQ2: Expedited Development Process through Feature Identification and Reuse**

  Feature modeling is an established software product line approach demonstrating the practical and time-saving aspects of reusing legacy code and core assets. The feature model includes already tested or verified features, facilitating faster software development. Addressing *RQ2*, the constructed feature model can significantly reduce the feature search time along with its valid configuration. The feature model's efficiency was highlighted by comparing feature search times with manual methods, proving its effectiveness in large-scale development. As depicted in Figure 6.2, the feature search times in the smart elevator system, infotainment system, and health monitoring system projects were reduced by 59.0%, 49.8%, and 74.5%, respectively. This reduction in search time and feature localization highlights the efficiency of our feature identification approach in the faster development of next-generation embedded software.

- **RQ3: Efficacy of BERT and Topic Modeling Techniques in n identifying and extracting features and requirements**

  To address Research Question *RQ3* on the efficacy of BERT and topic modeling in identifying features from embedded software, our investigations involved Electric Water Heaters, Microwaves, and Autonomous Vehicle Systems (AVS). Table 3.1 showcases the identified features and requirements for the Electric Water Heater and Microwave, using both LDA-based TF-IDF and BERT methods. BERT's effectiveness is further highlighted in Table 3.2 where BERT identified 29 features compared to 17 by LDA-based TF-IDF, indicating a more comprehensive feature extraction for Electric Water Heater project.

Figure 3.7 illustrates the feature coherence evaluation, demonstrating TF-IDF's precision in feature extraction. BERT's nuanced identification capabilities are evident in Figure 3.8, showcasing the similarities between various features and requirements for the Electric Water Heater. A comparative analysis of BERT and LDA across metrics like Coverage, Relevance, Granularity, and Diversity is depicted in Figure 3.9. BERT excels in Relevance and Granularity, while LDA shows better Coverage and Diversity.

For the AVS project, our BERT-enabled approach identified approximately 65 features, achieving an F1-score of 0.89, which notably surpassed the scores from manual and rule-based methods (Figure 3.11). The feature model for the AVS's Object Detection System (Figure 3.12) further confirms the intricate structure and interdependencies of features, offering valuable insights for efficient software development. Our findings confirm that BERT and topic modeling techniques are highly effective in extracting features and requirements from embedded software, providing detailed and context-specific insights that improve the extraction process significantly.

- **RQ4: Performance of Fog-Enabled Architecture**

  To address *RQ4* on the performance of fog-enabled architecture in satisfying embedded software constraints, our experimental testbed, depicted in Figures 6.3 and 6.4, played a crucial role. This setup effectively simulated a fog-assisted OTA update environment, allowing us to observe and evaluate the architecture's real-world capabilities. Notably, our testbed demonstrated significant improvements in OTA update times and network efficiency, vital in the context of embedded software systems.

  The comparative analysis of fog-based predicted OTA update times and handover delays against worst-case scenarios, shown in Figure 4.13, validated the effectiveness of our fog computing approach. These comparisons revealed that fog computing adeptly manages update times and handover delays, essential for maintaining seamless operation in embedded systems. The results from our testbed experiments provided a robust answer to *RQ4*. They confirmed that fog-enabled architecture effectively meets the diverse and often stringent constraints of embedded software, highlighting its potential as a leading solution for next-generation systems.

- **RQ5: Optimal Training of Large Machine Learning Models on Edge Network**

  RQ5 focused on addressing the challenge of training large ML models on edge devices by optimizing model partitioning. Our experiments with the AlexNet model (as detailed in Table 6.1 and Figure 6.6) showcased that strategic model partitioning leads to significant acceleration in training performance, maintaining accuracy. The experiment demonstrates an average speed-up of 2.5 times with our proposed pipelined model parallelism over sequential executions. In addition, we use the TMR technique for safe online model training on edge devices, which is discussed in Section 4.4.

6.6 Summary

This chapter addresses and clarifies the key research questions outlined in the Introduction chapter, focusing on next-generation embedded software development. Through a series of designed experiments and comprehensive analyses, we have validated our approaches across different facets of embedded software development, integrating advanced methodologies and modern computing paradigms.

- Investigating Legacy Embedded Software: The process of identifying and extracting reusable features from legacy embedded software is successfully executed using a combination of BERT and topic modeling techniques. The study on a GitHub project highlights our capability to determine functional and non-functional features from legacy systems.
- Integration of Fog Computing Architecture: Our experimental testbed underlined the effective performance of fog-enabled architecture in embedded systems. The OTA update times and network efficiency analysis confirm the dominance of fog computing in managing update times and handover delays, which is essential for next-generation embedded software systems.
- Optimizing ML Model Training on Fog/Edge Architecture: The experimental setup, involving the AlexNet model and outlined in Table 6.1 and Figure 6.6, demonstrate the importance of optimal model partitioning in accelerating machine learning model training on edge devices. This aspect of our study showcases the potential of integrating advanced machine learning models within embedded systems.

In summary, this chapter illustrates the applicability of our proposed approaches toward improving embedded software reuse. By integrating current computing techniques, such as fog and edge computing for large machine learning model training, into the framework of legacy software, we have demonstrated a clear pathway toward the evolution of next-generation embedded systems. These findings validate the potential of reusing legacy software and highlight the transformative role of modern methodologies in bridging traditional and contemporary software development practices.

Chapter 7

Conclusion and Future Work

7.1 Research Outcomes

The rapid evolution of embedded systems, particularly in the domain of Cyber-Physical Systems (CPS), has accentuated the need for agile software development methodologies. This research aimed to harness the potential of legacy software for developing reusable feature-based next-generation embedded systems. By integrating static analysis with topic modeling and the Bidirectional Encoder Representations from Transformers (BERT), the study identified reusable features from legacy software to expedite development and requirements analysis.

Key outcomes of the research include:

- **Feature and Requirement Analysis for Improving Reusability:** Developed a Python tool that automates the extraction of features from legacy software. This tool visually presents the requirements and constraints of features but also aids in constructing a comprehensive feature model. This innovation led to a significant 74.5% improvement in feature search times, enhancing the overall efficiency of the software development process.
- **A Framework for OTA Software Updates:** Successfully implemented and validated a fog computing testbed. This testbed showcases the efficacy of fog-assisted OTA updates in real-world scenarios, specifically focusing on vehicular software systems. The case study on vehicular OTA updates demonstrates the effectiveness of the proposed methodology in practical applications.
- **A Machine Learning Model Parallelism Tool:** Developed an advanced tool for optimizing machine learning model performance. This tool facilitates the partitioning of large ML models across edge

devices, ensuring optimal use of resources and improving computation efficiency. This approach is essential for integrating advanced functionalities of next-generation applications into feature-based embedded software.

7.2 Implications and Recommendations

This research presents advancements in embedded software development, setting the background for transformative changes in the industry. The methodologies and tools developed facilitate development and demonstrate the process of integrating cutting-edge technologies into embedded systems. This study offers insights and guidelines for current and future applications in this rapidly evolving field. The key implications and recommendations derived from our research are:

1. **Software Reusability:** Reusing legacy software can considerably reduce the time and cost of developing new software applications. This is especially beneficial in a competitive market where timely product releases are crucial.

2. **Enhanced Software Development:** With the ability to extract and visualize features and their requirements, developers can better understand software requirements, thus streamlining the design and development processes.

3. **Integration of Advanced Technologies:** The research emphasizes integrating advanced features like machine learning and fog computing to modernize embedded systems. Such integrations can lead to more efficient and reliable systems, especially within distributed environments.

Recommendations for industry and academia include:

- **Adopt the Feature-based Approach:** Organizations should consider adopting the proposed feature-based modular embedded software development approach, which can lead to faster, more efficient, and reliable software development.

- **Continuous Validation:** While the current validation has shown promising results, continuous testing and validation in diverse real-world scenarios are essential.

- **Training and Education:** As the approach integrates advanced tools and methodologies, organizations should invest in training and education to ensure that their teams can effectively use these methods.

7.3 Future Work

As we look ahead, the findings and methodologies of this research open up new scope in embedded software development. The potential for expansion and enhancement in various areas presents exciting opportunities for further innovation and refinement. These possibilities unfold current technologies and lay the groundwork for future demands. Here are some key areas of future work that stem from this study:

1. **Expand Language Scope:** While this research primarily focused on C-based software, there is potential to extend the methodology to other programming languages, broadening its applicability. We have shown how Python code of advanced features can be transformed into modular form to improve reusability.

2. **Improve the Tool:** The Python tool developed can be further improved, fine tuning CodeBERT, incorporating more features, better user interfaces, and integrations with popular development environments.

3. **Analyze More Machine Learning Algorithms:** As machine learning becomes more ingrained in embedded systems, there is scope to examine more optimizing models, ensuring their real-time performance, and exploring new algorithms that can be integrated.

4. **Scalability:** The current approach, while effective, needs to be tested for scalability, especially in large-scale embedded software projects. It would be worthwhile to explore how the methodology fares in larger, more complex environments.

5. **Integrating More Advanced Features:** Beyond fog computing and machine learning, future embedded systems might incorporate other advanced features. Research can explore the seamless integration of these features using the proposed approach.

In conclusion, this research has paved the way for a more agile, efficient, and feature-rich future in embedded software development. By leveraging the potential of legacy software, integrating advanced technologies, and focusing on a feature-based approach, the next-generation of embedded systems promises to be more robust, efficient, and aligned with the demands of the modern world.

Bibliography

[1] D. Fuller, “System design challenges for next generation wireless and embedded systems,” in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–1. DOI: 10.7873/DATE.2014.014.

[2] B. P. Douglass, *Design patterns for embedded systems in C: an embedded software engineering toolkit*. Elsevier, 2010.

[3] M. Al Maruf, A. Azim, and O. Alam, “Facilitating reuse of functions in embedded software,” *IEEE Access*, vol. 10, pp. 88 595–88 605, 2022.

[4] J. Krüger, W. Mahmood, and T. Berger, “Promote-pl: A round-trip engineering process model for adopting and evolving product lines,” in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, 2020, pp. 1–12.

[5] T. Eisenbarth, R. Koschke, and D. Simon, “Locating features in source code,” *IEEE Transactions on software engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[6] J. J. Labrosse, “Operating systems,” in *Software Engineering for Embedded Systems*, Elsevier, 2019, pp. 153–206.

[7] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yilmaz, “Testing embedded software: A survey of the literature,” *Information and Software Technology*, vol. 104, pp. 14–45, 2018.

[8] G. Xie, G. Zeng, Z. Li, R. Li, and K. Li, “Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems,” *IEEE Transactions on Vehicular Technology*, vol. 66, no. 8, pp. 6676–6692, 2017.

[9] P. Müller, K. Narasimhan, and M. Mezini, “Fex: Assisted identification of domain features from c programs,” in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2021, pp. 170–180.

[10] G. K. Michelon, L. Linsbauer, W. K. Assunccao, S. Fischer, and A. Egyed, “A hybrid feature location technique for re-engineeringsingle systems into software product lines,” in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2021, pp. 1–9.

[11] N. Islam and A. Azim, “Feature characterization for cps software reuse,” in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, 2019, pp. 314–315.

[12] T. Ziadi, L. Frias, M. A. A. da Silva, and M. Ziane, “Feature identification from the source code of product variants,” in *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, 2012, pp. 417–422.

[13] A. Burger and S. Gruner, “Finalist 2: Feature identification, localization, and tracing tool,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 532–537.

[14] R. Williams, T. Ren, L. De Carli, L. Lu, and G. Smith, “Guided feature identification and removal for resource-constrained firmware,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–25, 2021.

[15] N. Peinelt, D. Nguyen, and M. Liakata, “Tbert: Topic models and bert joining forces for semantic similarity detection,” in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 7047–7055.

[16] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, “Codebertscore: Evaluating code generation with pretrained models of code,” 2023. [Online]. Available: `https://arxiv.org/abs/2302.05527`.

[17] J. Delvin and M. Chang, *Open source bert: State-of-the art pre-training for natural language processing*, 2018.

[18] M. A. Maruf and A. Azim, “Automated features and requirements identification for improving cps software reuse using topic modeling,” in *Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023)*, 2023, pp. 262–263.

[19] D. Blouin, R. Al-Ali, M. Iacono, B. Tekinerdogan, and H. Giese, “An ontological foundation for multi-paradigm modelling for cyber-physical systems,” in *Multi-Paradigm Modelling Approaches for Cyber-Physical Systems*, Elsevier, 2021, pp. 9–43.

[20] L. Li, Y. Zheng, M. Yang, *et al.*, “A survey of feature modeling methods: Historical evolution and new development,” *Robotics and Computer-Integrated Manufacturing*, vol. 61, p. 101 851, 2020.

[21] D. Hinterreiter, K. Feichtinger, L. Linsbauer, H. Prahofer, and P. Grunbacher, “Supporting feature model evolution by lifting code-level dependencies: A research preview,” in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Springer, 2019, pp. 169–175.

[22] J. Shi, J. Bian, J. Richter, *et al.*, “Modes: Model-based optimization on distributed embedded systems,” *Machine Learning*, vol. 110, no. 6, pp. 1527–1547, 2021.

[23] D. L. Lewis, *Over-the-air vehicle systems updating and associate security protocols*, US Patent 9,464,905, Oct. 2016.

[24] D. A. Wood, *Chrysler pacifica uconnect problems cause lawsuit*, URL: https://www.carcomplaints.com/news/2019/chrysler-pacifica-uconnect-problems-lawsuit.shtml, Last Accessed: June 26, 2021. [Online].

[25] M. A. Maruf and A. Azim, “Automated features and requirements identification for improving cps software reuse using topic modeling,” in *Proceedings of the ACM/IEEE 14th*

International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023), ser. ICCPS '23, San Antonio, TX, USA: Association for Computing Machinery, 2023, pp. 262–263, ISBN: 9798400700361. DOI: 10.1145/3576841.3589626. [Online]. Available: https://doi.org/10.1145/3576841.3589626.

[26] M. Al Maruf, A. Azim, N. Auluck, and M. Sahi, "Towards safe online machine learning model training and inference on edge networks," in *2023 International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2023. DOI: 10.1109/ICMLA58977.2023.00161.

[27] M. A. Maruf and A. Azim, "Optimizing DNNs Model Partitioning for Enhanced Performance on Edge Devices," *Proceedings of the Canadian Conference on Artificial Intelligence*, Jun. 2023, https://caiac.pubpub.org/pub/ly32gqd5.

[28] A. Azim and M. Al Maruf, "Cognitive mobile computing for cyber-physical systems (cps)," in *Towards a Wireless Connected World: Achievements and New Technologies*, A.-S. K. Pathan, Ed. Cham: Springer International Publishing, 2022, pp. 203–222, ISBN: 978-3-031-04321-5. DOI: 10.1007/978-3-031-04321-5_9. [Online]. Available: https://doi.org/10.1007/978-3-031-04321-5_9.

[29] M. A. Maruf, A. Singh, A. Azim, and N. Auluck, "Faster fog computing based over-the-air vehicular updates: A transfer learning approach," *IEEE Transactions on Services Computing*, 2021.

[30] M. Al Maruf, A. Singh, A. Azim, and N. Auluck, "Resource efficient allocation of fog nodes for faster vehicular ota updates," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, IEEE, 2020, pp. 1–6.

[31] M. Majthoub, M. H. Qutqut, and Y. Odeh, "Software re-engineering: An overview," in *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, IEEE, 2018, pp. 266–270.

[32] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[33] Z. Jiang, A. El-Jaroudi, W. Hartmann, D. Karakos, and L. Zhao, “Cross-lingual information retrieval with bert,” *arXiv preprint arXiv:2004.13005*, 2020.

[34] M. Marques, J. Simmonds, P. O. Rossel, and M. C. Bastarrica, “Software product line evolution: A systematic literature review,” *Information and Software Technology*, vol. 105, pp. 190–208, 2019.

[35] D. Brugali, “Software product line engineering for robotics,” *Software Engineering for Robotics*, pp. 1–28, 2021.

[36] J. M. Ferreira, S. R. Vergilio, and M. Quinaia, “Software product line testing based on feature model mutation,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 05, pp. 817–839, 2017.

[37] K. Czarnecki and A. Wasowski, “Feature diagrams and logics: There and back again,” in *11th International Software Product Line Conference (SPLC 2007)*, IEEE, 2007, pp. 23–34.

[38] R. Mamata and A. Azim, “Work-in-progress: A resource-aware optimization model for real-time systems analysis and design,” in *2022 International Conference on Embedded Software (EMSOFT)*, IEEE, 2022, pp. 9–10.

[39] A. Larkin, *Disadvantages of Cloud Computing*. Cloud Adoption, Cloud Academy, Last accessed: November 16 2021, https://cloudacademy.com/blog/disadvantages-of-cloud-computing/, 2019.

[40] M. A. Maruf and A. Azim, “Requirements-preserving design automation for multiprocessor embedded system applications,” *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, pp. 821–833, 2021.

[41] Y.-H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "Rt-wifi: Real-time high-speed communication protocol for wireless cyber-physical control applications," in *2013 IEEE 34th Real-Time Systems Symposium*, IEEE, 2013, pp. 140–149.

[42] R. Hilbrich, "How to safely integrate multiple applications on embedded many-core systems by applying the "correctness by construction" principle," *Advances in Software Engineering*, 2012.

[43] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: Opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 10, pp. 5095–5115, 2022. DOI: 10.1109/TNNLS.2021.3071762.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[45] T. V. Renuka and B. Surekha, "Acute-lymphoblastic leukemia detection through deep transfer learning approach of neural network," in *Proceeding of First Doctoral Symposium on Natural Computing Research*, Springer, 2021, pp. 163–170.

[46] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[47] Y. Huang, Y. Cheng, A. Bapna, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[48] M. Abadi, P. Barham, J. Chen, *et al.*, "{Tensorflow}: A system for {large-scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[49] B. Hasheminezhad, S. Shirzad, N. Wu, P. Diehl, H. Schulz, and H. Kaiser, "Towards a scalable and distributed infrastructure for deep learning applications," in *In IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)*, 2020, pp. 20–30.

[50] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 101–110.

[51] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ecco tool: Extraction and composition for clone-and-own," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 2, 2015, pp. 665–668.

[52] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2013, pp. 604–621.

[53] M. Pfannemuller, C. Krupitzer, M. Weckesser, and C. Becker, "A dynamic software product line approach for adaptation planning in autonomic computing systems," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, IEEE, 2017, pp. 247–254.

[54] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, and A. Amjad, "A model driven reverse engineering framework for generating high level uml models from java source code," *IEEE access*, vol. 7, pp. 158 931–158 950, 2019.

[55] E. Kang and D. Jackson, "A model-based framework for security configuration analysis," *Unpublished manuscript. Available at: http://people. csail. mit. edu/eskang/papers/security-configuration. pdf*,

[56] J. Carbonnel, M. Huchard, and C. Nebut, "Towards the extraction of variability information to assist variability modelling of complex product lines," in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, 2018, pp. 113–120.

[57] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. Le Traon, "Automating the extraction of model-based software product lines from model variants (t)," in *2015 30th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 396–406.

[58] Y. Tang and H. Leung, "Sticprob: A novel feature mining approach using conditional probability," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2017, pp. 45–55.

[59] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering-Volume 1*, 2010, pp. 95–104.

[60] F. Perez, R. Lapena, A. C. Marcen, and C. Cetina, "Topic modeling for feature location in software models: Studying both code generation and interpreted models," *Information and Software Technology*, vol. 140, p. 106 676, 2021.

[61] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 529–533.

[62] M. S. Haris, T. A. Kurniawan, and F. Ramdani, "Automated features extraction from software requirements specification (srs) documents as the basis of software product line (spl) engineering," *Journal of Information Technology and Computer Science*, vol. 5, no. 3, pp. 279–292, 2020.

[63] A. E. Yaacoub, L. Mottola, T. Voigt, and P. Rümmer, "Timing analysis of embedded software updates," *arXiv preprint arXiv:2304.14213*, 2023.

[64] M. Azanza, A. Irastorza, R. Medeiros, and O. Diaz, "Onboarding in software product lines: Concept maps as welcome guides," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, IEEE, 2021, pp. 122–133.

[65] T. Bhowmik, N. Niu, A. Mahmoud, and J. Savolainen, “Automated support for combinational creativity in requirements engineering,” in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, IEEE, 2014, pp. 243–252.

[66] J. Kruger, T. Berger, and T. Leich, “Features and how to find them: A survey of manual feature location,” *Software Engineering for Variability Intensive Systems*, pp. 153–172, 2019.

[67] R. Li, L. B. Allal, Y. Zi, *et al.*, “Starcoder: May the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[68] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki, “Mining configuration constraints: Static analyses and empirical results,” in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 140–151.

[69] R. Damaševičius, P. Paškevičius, E. Karčiauskas, and R. Marcinkevičius, “Automatic extraction of features and generation of feature models from java programs,” *Information Technology and Control*, vol. 41, no. 4, pp. 376–384, 2012.

[70] M. Steger, C. A. Boano, T. Niedermayr, *et al.*, “An efficient and secure automotive wireless software update framework,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2181–2193, 2018.

[71] T. Mirfakhraie, G. Vitor, and K. Grogan, “Applicable protocol for updating firmware of automotive hvac electronic control units (ecus) over the air,” in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 21–26.

[72] S. Ingle and M. Phute, “Tesla autopilot: Semi autonomous driving, an uptick for future autonomy,” *International Research Journal of Engineering and Technology*, vol. 3, no. 9, 2016.

[73] S. Nie, L. Liu, Y. Du, and W. Zhang, "Over-the-air: How we remotely compromised the gateway, bcm, and autopilot ecus of tesla cars," *BlackHat USA, Las Vegas, NV*, pp. 1–19, 2018.

[74] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, 2018.

[75] Z. Zhou, P. Liu, J. Feng, Y. Zhang, S. Mumtaz, and J. Rodriguez, "Computation resource allocation and task assignment optimization in vehicular fog computing: A contract-matching approach," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 3113–3125, 2019.

[76] S. Memon and M. Maheswaran, "Using machine learning for handover optimization in vehicular fog computing," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ACM, 2019, pp. 182–190.

[77] S. Wang, J. Wu, S. Zhang, and K. Wang, "Ssds: A smart software-defined security mechanism for vehicle-to-grid using transfer learning," *IEEE Access*, vol. 6, pp. 63 967–63 975, 2018.

[78] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3702–3712, 2016.

[79] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, vol. 154, pp. 22–36, 2019.

[80] M. Maheswaran, T. Yang, and S. Memon, "A fog computing framework for autonomous driving assist: Architecture, experiments, and challenges," *arXiv preprint arXiv:1907.09454*, 2019.

[81] A. A. Khan, M. Abolhasan, and W. Ni, "5g next generation vanets using sdn and fog computing framework," in *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, IEEE, 2018, pp. 1–6.

[82] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *Journal of Systems and Software*, p. 111 351, 2022.

[83] A. Vladyko, A. Khakimov, A. Muthanna, A. A. Ateya, and A. Koucheryavy, "Distributed edge computing to assist ultra-low-latency vanet applications," *Future Internet*, vol. 11, no. 6, p. 128, 2019.

[84] J. Zhu, J. Hu, M. Zhang, Y. Chen, and S. Bi, "A fog computing model for implementing motion guide to visually impaired," *Simulation Modelling Practice and Theory*, vol. 101, p. 102 015, 2020.

[85] N. Suryadevara, A. Negi, and S. R. Rudraraju, "A smart home assistive living framework using fog computing for audio and lighting stimulation," in *Advances in Decision Sciences, Image Processing, Security and Computer Vision*, Springer, 2020, pp. 366–375.

[86] D. N. Jha, K. Alwasel, A. Alshoshan, *et al.*, "Iotsim-edge: A simulation framework for modeling the behavior of internet of things and edge computing environments," *Software: Practice and Experience*, vol. 50, no. 6, pp. 844–867, 2020.

[87] G. Kim and I. Y. J. and, "Integrity assurance of ota software update in smart vehicles," *International Journal on Smart Sensing and Intelligent Systems*, vol. 12, no. 1178-5608, pp. 1–8, 2019. DOI: 10.21307/ijssis-2019-011.

[88] Y. Bi, G. Han, C. Lin, Q. Deng, L. Guo, and F. Li, "Mobility support for fog computing: An sdn approach," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 53–59, 2018.

[89] T. Karthik, A. Brown, S. Awwad, *et al.*, "Uptane: Securing software updates for automobiles," in *International Conference on Embedded Security in Car*, 2016, pp. 1–11.

[90] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, “Memory-efficient pipeline-parallel dnn training,” in *International Conference on Machine Learning*, PMLR, 2021, pp. 7937–7947.

[91] D. Narayanan, A. Harlap, A. Phanishayee, *et al.*, “Pipedream: Generalized pipeline parallelism for dnn training,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[92] M. Tanaka, K. Taura, T. Hanawa, and K. Torisawa, “Automatic graph partitioning for very large-scale deep learning,” in *In IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 1004–1013.

[93] A. A. Moreno, J. Olivito, J. Resano, and H. Mecha, “Analysis of a pipelined architecture for sparse dnns on embedded systems,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1993–2003, 2020.

[94] D. Saguil and A. Azim, “A layer-partitioning approach for faster execution of neural network-based embedded applications in edge networks,” *IEEE Access*, vol. 8, pp. 59 456–59 469, 2020.

[95] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, “Adaptive parallel execution of deep neural networks on heterogeneous edge devices,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 195–208.

[96] A. Harlap, D. Narayanan, A. Phanishayee, *et al.*, “Pipedream: Fast and efficient pipeline parallel dnn training,” *arXiv preprint arXiv:1806.03377*, 2018.

[97] S. Fan, Y. Rong, C. Meng, *et al.*, “Dapple: A pipelined data parallel approach for training large models,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 431–445.

[98] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "Pipemare: Asynchronous pipeline parallel dnn training," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 269–296, 2021.

[99] V. Kianzad and S. S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, 2006.

[100] H.-I. Wu, D.-Y. Guo, H.-H. Chin, and R.-S. Tsay, "A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 46–49. DOI: 10.1109/AICAS48895.2020.9073977.

[101] D. Jeong, J. Kim, M.-L. Oldja, and S. Ha, "Parallel scheduling of multiple sdf graphs onto heterogeneous processors," *IEEE Access*, vol. 9, pp. 20 493–20 507, 2021.

[102] L. E. Rubio-Anguiano, A. C. Trabanco, J. L. B. Velasco, and A. Ramirez-Trevino, "Maximizing utilization and minimizing migration in thermal-aware energy-efficient real-time multiprocessor scheduling," *IEEE Access*, vol. 9, pp. 83 309–83 328, 2021.

[103] E. A. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors*, vol. 15, no. 3, pp. 4837–4869, 2015.

[104] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th design automation conference*, 2010, pp. 731–736.

[105] Z. Sharafi, Z. Soh, and Y.-G. Gueheneuc, "A systematic literature review on the usage of eye-tracking in software engineering," *Information and Software Technology*, vol. 67, pp. 79–107, 2015.

[106] I. Alazzam, A. Aleroud, Z. Al Latifah, and G. Karabatis, "Automatic bug triage in software systems using graph neighborhood relations for feature augmentation," *IEEE Transactions*

on Computational Social Systems, vol. 7, no. 5, pp. 1288–1303, 2020. DOI: 10.1109/TCSS.2020.3017501.

[107] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *2012 9th IEEE working conference on mining software repositories (MSR)*, IEEE, 2012, pp. 189–198.

[108] D. Yu and B. Xiang, "Discovering topics and trends in the field of artificial intelligence: Using lda topic modeling," *Expert Systems with Applications*, p. 120 114, 2023.

[109] P. Wu, J. Wang, and B. Tian, "Software homology detection with software motifs based on function-call graph," *IEEE Access*, vol. 6, pp. 19 007–19 017, 2018.

[110] E. Bendersky, *Pycparser. github repository 2020*, URL: https://github.com/eliben/pycparser.

[111] Z. Feng, D. Guo, D. Tang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[112] A. Elbanawi, *Electric-water-heater*, URL: https://github.com/AhmedElbanawi/Electric-Water-Heater.git, Accessed: 2023-0-20, 2020.

[113] A. Elbanawi, *Microwave*, URL: https://github.com/AhmedElbanawi/MicroWave.git, Accessed: 2023-07-12, 2020.

[114] M. A. Maruf, *Autonomous vehicle systems*, URL: https://doi.org/10.6084/m9.figshare.22237105, 2023.

[115] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog computing: A viewpoint of vehicles as the infrastructures," *IEEE Transactions on Vehicular Technology*, vol. vol. 65, no. 6, pp. 3860–3873, 2016.

[116] S. Halder, A. Ghosal, and M. Conti, "Secure ota software updates in connected vehicles: A survey," *arXiv preprint arXiv:1904.00685*, 2019.

[117] D. Kim, S. Kim, and J. H. Park, “Remote software update in trusted connection of long range iot networking integrated with mobile edge cloud,” *IEEE Access*, vol. 6, pp. 66 831–66 840, 2017.

[118] X. Jin, *A survey on network architectures for mobility*, 2006.

[119] G. Hampel, K. L. Clarkson, J. D. Hobby, and P. A. Polakos, “The tradeoff between coverage and capacity in dynamic optimization of 3g cellular networks,” in *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484)*, IEEE, vol. 2, 2003, pp. 927–932.

[120] M. Kane, “Mass over-the-air update of tesla cars captured on video,” *InsideEVs (Electric Vehicle News, Reviews, and Reports)*, Last Accessed: June 27, 2021.

[121] K. Weiss, T. M. Khoshgoftaar, and D. Wang, “A survey of transfer learning,” *Journal of Big Data*, 2016. DOI: https://doi.org/10.1186/s40537-016-0043-6.

[122] P. Chawdhry, G. Folloni, S. Luzardi, and S. Lumachi, *European wifi hotspot signal strength coverage*, 2016. [Online]. Available: http://data.europa.eu/89h/jrc-netbravo-netbravo-od-eu-wifi.

[123] D. A. King and J. F. Saldarriaga, “Access to taxicabs for unbanked households: An exploratory analysis in new york city,” *Journal of Public Transportation*, vol. 20, no. 1, p. 1, 2017.

[124] M. A. Maruf and A. Azim, “Extending resources for avoiding overloads of mixed-criticality tasks in cyber-physical systems,” *IET Cyber-Physical Systems: Theory & Applications*, 2019.

[125] D. Raca, D. Leahy, C. J. Sreenan, and J. J. Quinlan, “Beyond throughput, the next generation: A 5g dataset with channel and context metrics,” in *Proceedings of the 11th ACM Multimedia Systems Conference*, ser. MMSys ’20, Istanbul, Turkey: Association for Computing

Machinery, 2020, pp. 303–308, ISBN: 9781450368452. DOI: 10.1145/3339825.3394938. [Online]. Available: https://doi.org/10.1145/3339825.3394938.

[126] X. Chen, A. Azim, X. Liu, S. Fischmeister, and J. Ma, "Dts: Dynamic tdma scheduling for networked control systems," *Journal of Systems Architecture*, vol. 60, no. 2, pp. 194–205, 2014.

[127] X. Wang, X. Wu, S. Cheng, J. Shi, X. Ping, and W. Yue, "Design and experiment of control architecture and adaptive dual-loop controller for brake-by-wire system with an electric booster," *IEEE Transactions on Transportation Electrification*, vol. 6, no. 3, pp. 1236–1252, 2020.

[128] K. Cao, S. Hu, Y. Shi, A. W. Colombo, S. Karnouskos, and X. Li, "A survey on edge and edge-cloud computing assisted cyber-physical systems," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 11, pp. 7806–7819, 2021.

[129] M. Al Maruf, A. Singh, A. Azim, and N. Auluck, "Faster fog computing based over-the-air vehicular updates: A transfer learning approach," *IEEE Transactions on Services Computing*, vol. 15, no. 6, pp. 3245–3259, 2022.

[130] M. Jouhari, A. K. Al-Ali, E. Baccour, *et al.*, "Distributed cnn inference on resource-constrained uavs for surveillance systems: Design and optimization," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1227–1242, 2021.

[131] J. Na, H. Zhang, J. Lian, and B. Zhang, "Partitioning dnns for optimizing distributed inference performance on cooperative edge devices: A genetic algorithm approach," *Applied Sciences*, vol. 12, no. 20, p. 10 619, 2022.

[132] M. Shahshahani, P. Goswami, and D. Bhatia, "Memory optimization techniques for fpga based cnn implementations," in *2018 IEEE 13th Dallas Circuits and Systems Conference (DCAS)*, IEEE, 2018, pp. 1–6.

[133] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[134] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[135] A. Menshchikov, D. Shadrin, V. Prutyanov, *et al.*, "Real-time detection of hogweed: Uav platform empowered by deep learning," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1175–1188, 2021.

[136] I. Z. Mukti and D. Biswas, "Transfer learning based plant diseases detection using resnet50," in *2019 4th International conference on electrical information and communication technology (EICT)*, IEEE, 2019, pp. 1–6.

[137] P. Moritz, R. Nishihara, S. Wang, *et al.*, "Ray: A distributed framework for emerging {ai} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[138] J. M. G. Sánchez, N. Jörgensen, and M. Törngren, "Edge computing for cyber-physical systems," *ACM Trans. Cyber-Phys. Syst.*, 2022.

[139] S. Hamdan, S. Almajali, M. Ayyash, H. B. Salameh, and Y. Jararweh, "An intelligent edge-enabled distributed multi-task learning architecture for large-scale iot-based cyber–physical systems," *Simulation Modelling Practice and Theory*, vol. 122, p. 102 685, 2023.

[140] W. Sun, J. Liu, and Y. Yue, "Ai-enhanced offloading in edge computing: When machine learning meets industrial iot," *IEEE Network*, vol. 33, no. 5, pp. 68–74, 2019.

[141] G. Zhu, D. Liu, Y. Du, C. You, J. Zhang, and K. Huang, "Toward an intelligent edge: Wireless communication meets machine learning," *IEEE communications magazine*, vol. 58, no. 1, pp. 19–25, 2020.

[142] S. Shahhosseini, D. Seo, A. Kanduri, *et al.*, "Online learning for orchestration of inference in multi-user end-edge-cloud networks," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 6, pp. 1–25, 2022.

[143] T. Arifeen, A. S. Hassan, and J. Lee, "Approximate triple modular redundancy: A survey," *IEEE Access*, vol. 8, pp. 139 851–139 867, 2020.

[144] Kaggle, *Traffic, driving style, and road surface condition*, Available online: https://www.kaggle.com/datasets/ driving-style-road-surface-condition (accessed on 20 March 2023), 2023.

[145] M. Ruta, F. Scioscia, G. Loseto, A. Pinto, and E. Di Sciascio, "Machine learning in the internet of things: A semantic-enhanced approach," *Semantic Web*, vol. 10, no. 1, pp. 183–204, 2019.

[146] R. Tong, Q. Jiang, Z. Zou, T. Hu, and T. Li, "Embedded system vehicle based on multi-sensor fusion," *IEEE Access*, 2023.

[147] V. Kamath and A. Renuka, "Deep learning based object detection for resource constrained devices-systematic review, future trends and challenges ahead," *Neurocomputing*, 2023.

[148] J. Lee, M. Stanley, A. Spanias, and C. Tepedelenlioglu, "Integrating machine learning in embedded sensor systems for internet-of-things applications," in *2016 IEEE international symposium on signal processing and information technology (ISSPIT)*, IEEE, 2016, pp. 290–294.

[149] A. Ballesteros, M. Barranco, J. Proenza, L. Almeida, F. Pozo, and P. Palmer-Rodriguez, "An infrastructure for enabling dynamic fault tolerance in highly-reliable adaptive distributed embedded systems based on switched ethernet," *Sensors*, vol. 22, no. 18, p. 7099, 2022.

[150] J. D. S. Eleuterio, B. B. de Francca, C. M. Rubira, and R. de Lemos, "Realising variability in dynamic software product lines," in *Software Engineering for Variability Intensive Systems*, Auerbach Publications, 2019, pp. 195–223.

[151] O. Toolkit, *Face detection mtcnn python* demo*, https://github.com/openvinotoolkit/open_model_zoo/tree/master/demos/face_detection_mtcnn_demo, Accessed: 2023-07-12, 2023.

[152] U. T. Pedro Coser, *Software controller for vehicles . github repository 2020*, URL: https://github.com/PCoser/Software-Programing-in-C.

[153] J. Liang, *Elevator simulation. github repository 2016*, URL: https://github.com/JustinLiang/ElevatorSimulation.

[154] Y. Malinov, *Car infotainment. github repository 2015*, URL: https://github.com/YMalinov/car-infotainment.git.

[155] vikrant thakur, *Health monitoring system. github repository 2020*, URL: https://github.com/vikrantdeveloper/health-monitoring-system.

[156] *R-Car H3 High-end Automotive System-on-Chip (SoC) for In-vehicle Infotainment and Driving Safety Support, Renesas Electronics*, URL: https://www.renesas.com/us/en/application/automotive/r-car-h3-m3-h2-m2-e2-documents-software, Last accessed: April 20, 2021.

[157] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

Appendices

Appendix A

List of Symbols

- X denotes a given code segment.
- $\mathrm{G(X)} = \{\phi_1(\mathrm{X}), \phi_2(\mathrm{X}), ..., \phi_\mathrm{n}(\mathrm{X})\}$ represents the function call graph of code segment X.
- $\phi_\mathrm{i}(\mathrm{X})$ denotes a specific function and its interactions within the code snippet X.
- $\mathrm{f}_1, \mathrm{f}_2$ are features within a feature model, subject to requires and excludes constraints.
- $\mathrm{F(X)} = \{\mathrm{f}_1, \mathrm{f}_2, ..., \mathrm{f}_\mathrm{k}\}$ is a vector representing features of a code segment X.
- $\mathrm{R(X)} = \{\oplus \mathrm{r}_{\mathrm{f1}}, \oplus \mathrm{r}_{\mathrm{f2}}, \otimes \mathrm{r}_{\mathrm{nf1}}, \otimes \mathrm{r}_{\mathrm{nf2}}, ...\}$ is a vector representing functional and non-functional requirements of X.
- $\oplus$ symbolizes functional requirements.
- $\otimes$ symbolizes non-functional requirements.
- $\mathrm{R}_\mathrm{f}, \mathrm{R}_\mathrm{nf}$ categorize requirements into functional and non-functional types.
- $\mathrm{M(F)}, \mathrm{M'(F)}$ are mappings from features to functional and non-functional requirements respectively.
- F^in is a matrix defining the number of incoming function calls for each feature.
- d_ij stores the degree of incoming function calls to feature f_i from f_j.
- V, E in graph $G = (V, E)$ represent the set of all functions and the interactions between these functions respectively.
- D is a matrix where each element d_{ij} shows the type of relation between functions v_i and v_j.
- $\mathrm{G}(\phi)$ assigns functions ϕ to groups based on their role within the software.

- A is an artifact representing a segment of source code.
- T is a set of tokens obtained from artifact A.
- $\mathrm{T'}$ represents non-essential tokens removed during preprocessing.
- $\mathrm{T_{clean}}$ is the refined set of tokens after cleaning.
- $\mathrm{LCU(A_i, A_j)}$ identifies the Longest Common Unit between artifacts A_i and A_j.
- V is the Bag-of-Words vector for any artifact A.
- $\mathrm{f(t_i)}$ indicates the occurrence count of token t_i in A.
- $\mathrm{TF - IDF(t_i, A)}$ represents the Term Frequency-Inverse Document Frequency weighting for token t_i in artifact A.
- $\mathcal{F}$ is a set of features in a feature model.
- $\mathcal{M}$ is a feature model representing relationships among features in $\mathcal{F}$.
- $\mathrm{\Delta_i}$ denotes the total handover delay for the i^{th} vehicle.
- $\mathrm{D_{rang}}$ represents the time required for the initial ranging process to find fog nodes with maximum coverage.
- $\mathrm{D_{req}}$ is the time required to request a connection to a new fog node.
- $\mathrm{D_{res}}$ indicates the time required to register with the target fog node.
- $\mathrm{D_{ex}}$ is the time required for message exchange during the handover process.
- β is a weighting factor emphasizing the importance of communication latency.
- γ is a weighting factor balancing the significance of TMR time in optimization.
- $\mathrm{s_k}$ denotes a set of partitions for edge devices.
- $\mathrm{t_{m_i}}$ represents the training time for model split m_i.
- $\mathrm{l_{m_i}}$ denotes the communication latency for model split m_i.
- $\mathrm{T_{TMR_{k,m_i}}}$ is the TMR time for partition k of model split m_i. .
- $\mathrm{C_i}$ denotes Cluster number i; where $1 \leq i \leq m$; $m \in \mathbb{N}$.

- n represents the number of fog nodes.
- $\mathrm{F_k}$ is the k^{th} fog node where $1 \leq k \leq n; n \in \mathbb{N}$.
- $\mathrm{U_k^f}$ indicates the maximum resource utilization capacity of fog F_k.
- $\mathrm{C_i^{size}}$ is the size of Cluster C_i.
- $\mathrm{C_i^f}$ denotes the capacity of fog node F_i (requests/second).
- $\mathrm{N(C_i)}$ is the number of fog nodes in Cluster C_i.
- $\mathrm{N'(C_i)}$ refers to the number of active fog nodes in Cluster C_i.
- $\mathrm{V(C_i)}$ signifies the number of vehicle requests in Cluster C_i.
- $\mathrm{p_j}$ is the computational power required by vehicle V_j.
- $\mathrm{A(C_i)}$ is the number of active fog nodes required to handle $V(C_i)$.
- $\mathrm{S_j^{size}}$ is the software update size for vehicle V_j; $j \in \mathbb{N}$.
- $\mathrm{Pkt_j^{arrive}}$ is the j^{th} packet arrival time to the destination; $j \in \mathbb{N}$.
- $\mathrm{DR_j}$ is the network throughput for software size S_j^{size}.
- $\mathrm{Pkt_j^{send}}$ is the j^{th} packet sending time; $j \in \mathbb{N}$.
- $\mathrm{T_j}$ is the total time taken to update S_j^{size} for vehicle V_j.
- $\mathrm{cd_{ik}}$ is the communication distance between i^{th} vehicle and k^{th} fog node.
- $\mathrm{T_i^p}$ is the propagation/communication delay between i^{th} vehicle and k^{th} fog node.
- $\mathrm{T_i^t}$ represents the processing or transfer time.
- $\mathrm{T_i^{lt}}$ denotes the network latency.
- $\mathrm{Y_i}$ is the actual target value.
- $\mathrm{\Delta_i}$ indicates the handover delay.
- $\mathrm{Y_i'}$ is the predicted delay value.
- $\mathrm{W_i}$ represents the worst-case delay.
- $\mathrm{A_i^p}$ denotes the accuracy in the predicted model.

- $\mathrm{A}_{\mathrm{i}}^{\mathrm{P}}$ (second instance) represents the expected accuracy set by the engineer.
- $\mathbb{Z}+$ denotes positive integer numbers.

Appendix B

Extracted Features of Software Car Controller

F_id	Feature	Type / Relation	Requires / Excludes	Variables and Constraints	Sub-features
1	System	Mandatory		V:{vehicle_status}	{bus_controller, airbag, infotainment, engine, sensors,actuators, engine, gear_shift, light_controller}
2	Test	Optional			{software, hardware}
3	bus_controller	Mandatory		V:{gas_pedal_pos, brake_pedal_pos, Comm_bus_message }	
4	airbag	Mandatory		V:{crash_threshold, inflate_speed} CL:{crash_threshold=10mph, inflate_speed=100mph}	
5	infotainment	Mandatory		V:{ICE_WARNING_DELAY}	{display, communication}
6	engine	Mandatory		V:{Temp_refresh_interval} CL:{Temp_refresh_interval=3000ul}	{gasolin, electric}
7	gear_shift	Mandatory		V:{eLimitedSpeed, minRPMs, redline}} CL:{ Gear(0, 0, 0, true, eLimitedSpeed, minRPMs, redline), Gear(1, 35, 0, false, eLimitedSpeed, minRPMs, redline), Gear(2, 65, 8, false, eLimitedSpeed, minRPMs, redline), Gear(3, 95, 30, false, eLimitedSpeed, minRPMs, redline), Gear(4, 110, 45, false, eLimitedSpeed, minRPMs, redline), Gear(5, 125, 60, false, eLimitedSpeed, minRPMs, redline), Gear(6, 155, 85, false, eLimitedSpeed, minRPMs, redline), Gear(-1, -30, 0, false, eLimitedSpeed, minRPMs, redline) }	{manual, automatic}
10	actuators	Mandatory		V:{brake_actuator_pos, fuel_actuator_pos, direction_actuator_pos} CL: {event_action(time, period)}	{direction, fuel, brake,}
11	sensors	Mandatory		V: {vehicle_wheel_rotation, detection_time} CL: {collect_data()}	{gas_pedal, brake_pedal, steering_wheel, wheel_sensor}
12	light_controller	Mandatory		V:{light_status, light_pos}	{sidelight, low_beam, high_beam, auto_set}
13	gas_pedal	Mandatory		V:{gas_pedal_pos} CL:{change_speed(speed)}	
14	brake_pedal	Mandatory		V:{brake_pedal_pos}	
15	steering_wheel	Mandatory		V:{steering_wheel_pos}	
16	wheel	Mandatory		V:{avgWheelPeriod, wheelMode, wheel_angle} CL:{lockUpValue(), , wheel_rotation_count()}	
17	manual	Alternative	{Excludes: automatic}		
18	automatic	Alternative	{Excludes: manual}		
19	direction	Mandatory			
20	fuel	Mandatory		V:{fuel_actuator_pos}	
21	brake	Mandatory		V: {ADC_brake_value, brake_mode} CL:{EMERGENCY_STOP=1000, MIN_BRAKE=10, MIN_PERIOD=60, ADC_FREQUENCY=50}	
22	gasoline	Mandatory			
23	electric	Optional			
24	sidelight	Mandatory			
25	low_beam	Mandatory			
26	high_beam	Mandatory			
27	auto_set	Equivalent	{Requires: sidelight, low_beam, high_beam}		
28	vehicle_velocity	Optional	{Requires: wheel}	V:{wheel_rotation, distance}	
29	communication	Mandatory		V:{comm_type}	{gps, weather}
38	weather	OR		V:{location, weather_status}	

Table B.1: Extracted features from GitHub project “Software Controller for Vehicles” [152].