The architectural erosion problem in Andorid Apps

Identifying Architectural Erosion in Android Apps

Juan Camilo Acosta Rojas

October 3, 2025

This thesis is submitted in partial fulfillment of the requirements for a degree of Master in Systems and Computing Engineering (MISIS).

Thesis Committee:

Prof. Name Last name (Promotor)	Universidad de los Andes, Colombia
Prof. Reviewer 1 Reviewer 1	Institution, Country
Prof. Reviewer 2 Reviewer 2	Institution, Country

Identifying Architectural Erosion in Android Apps

The Software Design Lab

Systems and Computing Engineering Department

Faculty of Engineering

Universidad de los Andes

Bogotá, Colombia

To my family. The fruit of all their efforts be shown here.

Abstract

Software engineering involves different efforts with the same purpose: high-quality software development. Among these efforts, we can find the software architecture definition process, which gives preference to the various quality attributes according to the system's needs. However, for different reasons, the solution development can deviate from the original architecture design, which can generate performance problems and, consequently, affect the user experience. This impact on software quality could be represented, at a greater rate, in a mobile environment due to its limited resources. Previous research with this approach has focused on problem study and detection in different areas, like security, connectivity, etc. However, in mobile development, the concept of "architectural erosion" has not been deeply studied. The primary objective of this research project is to identify and locate architectural erosion bugs in mobile applications, utilizing two solution approaches: static analysis code techniques and the application of AI models and NLP fundamentals in commit analysis of Android projects. The results of each methodology will be discussed and extended to solve some issues in Android projects.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

At the beginning of system building, we might undergo a strong design process, where we define the system components and resources that we need to use. This process is named *software architecture definition*. After that, you have to align the development process with those architectural rules to achieve goals in performance, security, availability, and others. However, due to growth in software development, as time goes by, software tends to violate architectural rules, affecting system quality attributes. This could generate issues in security, availability, performance, and latency. This deviation is called *architectural erosion*. In mobile development, it can reduce application performance and other system quality attributes, affecting device resources and, ultimately, the user experience. In recent research, there have been significant advances in bug resolution across different approaches. For example, in security, connectivity, and code smells, there are various tools and components that detect and offer solutions for different bugs related to these approaches.

However, in the architectural erosion analysis, despite some research tackling this concept, some of these propose a toolset; it is not clear how we can resolve architectural erosion insights or a set of directives for facing this problem in the mobile development ecosystem. For this reason, it is important to find the impact of architectural erosion in mobile applications and how we can identify and locate architectural erosion bugs with two proposed methodologies:

1.1. Why make research efforts in Architectural Erosion Identification?

Due to different research, the costs in terms of human resources, technological resources, time, and money would grow after the first release and deployment of the software project. By this, it is necessary to establish the different reasons and artifacts that affect the software sustainability, the software performance in the short, middle, and long term. For this task, many approaches analyze different software project components like its design, its code, and its architectural rules. One of the most important, and one of the most studied, too. It has been the static code analysis approach. In this approach, a lot of tools have been created to identify and detect architectural erosion in specific code fragments, showing an analysis over time. These tools have been very helpful for the developers and for the teams, but it has not been enough in the mobile development ecosystem, where system resources are more limited than in the side-server environment. It is necessary to build a component with a specific purpose in Android Apps and show the importance of detecting architectural erosion in mobile app code and recovering these issues in an early way. Another approach is the identification with Natural Language Processing (NLP) techniques in commits of versioning platforms in a set of Android projects.

1.2. Machine learning models in architectural erosion

Recent research in Machine Learning models, specifically in Natural Language Processing, has concluded that the use of pre-trained models for the classification or generation of words in a specific language, their performance with programming languages is better than natural languages (in this NLP approach, English, Spanish, etc), according to their syntax. This idea could be recreated in issue detection in source code repository analysis. Issues like masked code, pattern design detection, and others have been implemented with pre-trained machine learning models. In architectural erosion, it could be useful for detecting the first symptoms of this and making early detection for, after that, using specific rules of architecture in static code analysis. In summary, machine learning models could detect the first symptoms of architectural erosion in an Android project and create a component with optimized performance for detecting and

recovering issues about this.

The objective of this research is to explore two identification methodologies of AER issues applied in Android projects. The first one is based on Static Code Analysis techniques. The main objective is to identify implemented work using the proposed techniques according to this methodology. The second one is based on the use of Artificial Intelligence (AI) models and NLP fundamentals. The objective of using this methodology is to analyze a large set of extracted commits from GitHub of a set of Android projects. The results of both methodologies are tested with judgments and comparisons of the results, and the use of AI techniques.

Furthermore, we will explore the alternatives of the use of these methodologies for solving more issues related to specific quality attributes in Android projects, like usability. We will extend the methodology based on NLP and AI models to explore the detection of usability issues in Android apps and the use of different tools based on Static code analysis to detect usability issues in Android projects. Finally, we explore the data analysis of reviews of Android apps as an extension to identify prominent issues based on the Applications and their reviews over time.

Chapter 2

Definitions

2.1. Development Process of a Software Project

When you have to make a software project plan, you have to define the main requirements that achieve the main goals of the business idea within the software project realization. First, you have to define the functional requirements. In software engineering, functional requirements define the main features of each software component involved in the project. Requirements like design, basic functionalities, and defined flows are built on the functional requirements definition process. After that, you have to define the main requirements that do not directly impact the user experience, which are called non-functional requirements. Non-functional requirements indirectly affect your software project; poor planning of these requirements can progressively impact user performance and the software project architecture, resulting in slower, heavier, or less useful applications for individual users. In this process, you must design a solution that satisfies the functional and non-functional requirements. This process is called the design process of a software project. Inside this stage, you must define the number of components that give the software its functionalities. Furthermore, you have to each component infrastructure into a process named Software Architecture Definition Laplante [5]. The software development process involves some steps:

2.1.1. Discovery

This stage consists of the definition of the main features of the software application. The discovery process defines the objective of the application and its main requirements. Furthermore, this gives an initial version of the behavior and the design of the software application.

2.1.2. Design

After determining the application requirements, it is possible to define the design of the software application's components and their relationships. In this stage, we define the functional and non-functional requirements. The functional requirements define the main features and use cases of the software application, defining a complete flow with the final user. The non-functional requirements define the performance metrics that the application must achieve. Based on the ISO rule, which defines the quality assurance of any software application. There is a set of defined non-functional requirements categories for achieving good performance in a software application: Scalability, availability, security, latency, integration capacity, and modification capacity (We will talk about these categories later). Once the prioritized non-functional requirements are defined, we define the rules and standards that will manage the performance of the software application.

2.1.3. Development

With the initial design of the software application and its components, we can implement and develop the software application. With a bad design process or a bad development process, it is possible to generate a deviation from the initial design of the application or a violation according to the initially defined standards of the software application.

2.1.4. Testing and Quality Assurance (QA)

With the design of the software application and its implementation, it is possible to test the main features and flows based on the designed use cases of the software application. In this stage, it would report code smells or bad performance metrics, according to the prioritized non-functional requirements.

2.1.5. Release

Once the application is tested, it achieves its performance metrics. The software application deploys in a production environment, with the final users' direct interaction. If the design process or the development process is implemented badly, it would affect the user performance of any software application. Focused on Android application development, the user experience would decay significantly, decreasing the number of active users or increasing the resources used in Android devices as the main consequences.

2.1.6. Maintenance

After the release and deployment stage. For a guarantee of the application's sustainability. It should create a refactoring process. That process consists of libraries updating, code smells fixing, and fixing bad design or development issues. The costs of one software application in terms of human resources, time, and money could increase the time goes by if the latest stages are implemented badly.

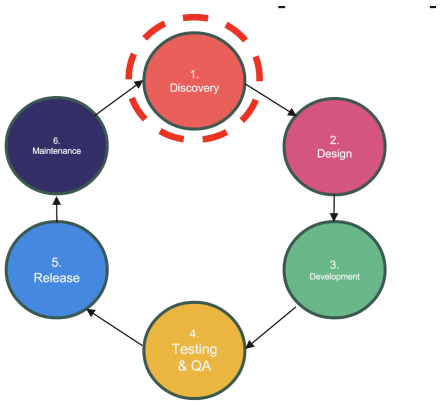

Figure 2.1: The development process of a Software Application

2.2. Architectural design in software engineering

For an effective and efficient software project development, we have to realize the respective investment in each stage. From the functional requirements creation to the design of each

component and its features and constraints. To do this, it is necessary to make an architectural design. The architecture of a software project. In general, defines the components that require the entire system and the connections and relationships (and their types). Furthermore, each component has to be defined in this design. In that order, different standards ensure the system's quality according to its architectural design.

2.2.1. Architectural design in Android development

Architectural Design in the mobile approach has been advancing step by step; this depends on the creation of new technologies in each development layer since the data management layer with the implementation of new database libraries like the room to the User Interface layers, with a different way of creating new screens inside the application (with the use of either JetPack Compose or Fragments organization). Depending on what the simple should be, an application, how many components it would have, which of them would be connected, and the reasons for those connections. It is necessary to review different architectural patterns used recently and why we focus on one of them for architectural erosion detection.

- MVC (Model View Controller): In this pattern, we divide the application components into the model, where we implement the connections with external platforms and internal data management. The controller component is used for setting the relation between the business logic and the User Interface (UI). The view component contains the UI. This pattern has been commonly used for the last 15 years due to its simplicity and popularity. However, the applications that use this pattern are very coupled, and their components depend strongly on others. It is usually found in business logic implementations and UI code fragments in the same code file, or data processing in business code components. At the beginning of Android, architectural issues weren´t as important as they are today. If an Android application is simple in terms of realization, it is possible to use an MVC pattern.

- MVP (Model View Presenter): This pattern is managed in a different way than the MVC pattern in the relations between business logic and UI. In this case, we use a component named presenter to manage the events and behavior of each UI view or screen. This pattern is commonly used for single Applications that do not have scalability or application

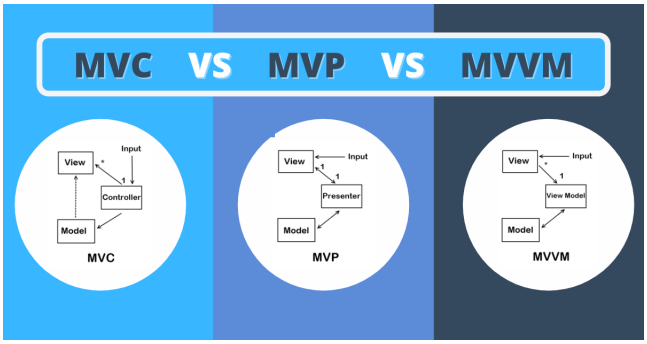

Figure 2.2: Similarities and differences between Android architectural patterns.]

overloading. This pattern divides the presenter features connected with the UI features. The disadvantages of this pattern are related to a high coupling rate inside its components and a high complexity for managing the life cycles of an application. Furthermore, it is difficult to implement new feature development and maintenance for large-scale Android applications.

- MVM (Model View View-Model): This pattern is one of the derivatives of clean architecture, a concept widely developed in Backend and Frontend applications architectures. This pattern uses some concepts of clean architecture, like use case organization, when we implement a new feature in an independent way from others. With this pattern, we use reactive components. The application components use libraries like Dagger Hilt to implement the use of reactive data; this reactive data changes depending on a UI event. With the creation of the JetPack Compose framework. The JetPack Compose is based on reactive UI and is more declarative than the traditional form (the use of XML files and fragments structure for managing different application screens)

In an Android application, it is not mandatory to use only an architectural pattern to achieve the functional and non-functional requirements of a software project. For example, it is very common to use the repository pattern declared in MVVM, divided into two: external connections and internal data management. Today, in the actual mobile development ecosystem, the most common framework that could be implemented with one or more architectural patterns is the three-layer pattern architecture: The UI layer, an optional layer named the Domain Layer,

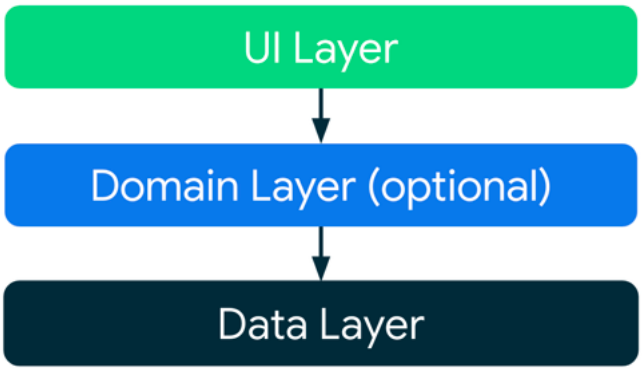

Figure 2.3: Three-layer framework for mobile development.]

and the Data layer. Each layer could be or could not be, depending on each application's requirements.

The three-layer architecture will be used as the main architectural framework to realize this study with the MVVM architectural pattern. These patterns are the most used in modern mobile development and will give a first approach for how developers can detect and fix architectural violations in an Android application.

2.2.2. Quality Attributes

The consequences of the generation of architectural erosion could impact the different quality attributes contemplated. According to Bass Bass et al. [2]. There are six attributes based on ISO-25010. Those attribute qualities are:

- Latency: This attribute measures the response time of the components according to the implemented architecture. Response times of each architectural component are very important.
- Scalability: Measures the ability of a system to grow in critical situations of system inputs. It is important for system availability when the user's connection rate increases.
- Security: Nowadays, a system must give protection to its users and their data, since the availability, integrity, and confidentiality of that data. This requirement is very important

in the legal environment of a system.

- Availability: When the system can be available when a system failure occurs (it doesn't depend on the failure type). This quality attribute measures the time that the system takes to recover from a failure.
- Integration capability: Measures the ability of a system to integrate with other(s) systems (s), measures the time and effort that the system needs to make that integration in all its layers, from the data layer to the UI layer is necessary.
- Modification capability: Similar to the last quality attribute, it measures the time that the system needs to change one or more components inside its system. The effort measure could be many relationships (human resources effort, time costs, money costs, etc.)

2.3. Architectural Erosion

The Architecture Erosion concept was treated a long time ago. Since 1992, architecture erosion has had a formal definition, giving the relationship of this concept with architectural rules violations Perry [8]. It can be treated from different perspectives, from the violation of rules, structure, and quality to the evolutionary perspective. Furthermore, the concept has a strong relationship with other synonyms, like, for example, degradation. In another modern definition, architectural erosion is defined as the set of architectural violations that reflect the deviation of the implemented architecture from the intended architecture over time. In summary, in a more concrete definition, architectural erosion could be considered a phenomenon that reflects the deviation of an implemented architecture from the intended architecture in a software project.

2.3.1. Approaches and Perspectives

Due to the original concept of architectural erosion Ruiyin et al. [9]. It could be studied and analyzed by different approaches:

- Violation perspective: Denotes how the implemented architecture violates the design principles or main constraints of the intended architecture. These violations could occur in

two phases: the design phase and the maintenance and evolution phase, making different changes step by step in the short and long term.

- Structure perspective: Where the structure of a software system encompasses its components and their relationships.
- Quality perspective: it refers to the degradation of the system quality, due to architectural changes that would generate architectural smells. It could include all the quality attributes contemplated in the industry.
- Evolution perspective: It shows the architectural inflexibility that increases the difficulty of implementing changes in the project and, therefore, decreases the sustainability of the system.

2.3.2. Main Reasons and Symptoms

Different factors could provoke architectural erosion in different stages of software project development. Due to these reasons, it is possible to make different solution approaches, and, therefore, the possibility of building different components based on those approaches. The main reasons found are:

- Architecture modularization: Due to the business needs, it is necessary to divide responsibilities between different components and layers in a software project. But sometimes it could produce non-functional components and deviate from the initial intended architecture.
- Architecture complexity: if the intended architecture is very complex, it is possible to deviate from it the time goes by, producing the first symptoms of architectural erosion.
- Architecture size: Due to this attribute, and with no control over the maintenance of the software project, it is not possible to have good software maintenance for a long time.
- Design Decisions: If the design decisions during the initial stages of the project don't have enough support (like documentation, reviews, etc.), it could generate problems with

the maintenance of refactoring of different components, decreasing the code quality and generating the first symptoms of architectural erosion.

- Duplicate functionality: As a consequence of the latest reasons too, duplicate functionality reflects the bad connection between layers of an architecture, and could be considered as the initial symptom of architectural erosion.

There are other reasons like bad documentation, bad programming features, but, in general, the main reasons were considered according to the architecture design stage issues.

2.3.3. Consequences

Are several points of view about the real definition of the consequences of architectural erosion violations inside a software project. The main problems that could generate the correct maintenance and the deviation from an intended architecture are:

- Costs of software maintenance: Due to the lack no implementation of architectural erosion issues, this could affect one of the non-functional defined requirements, and after that, affect the actual software infrastructure.
- Software Performance: When one of the quality attributes is affected by the initial architecture design planning, it incurs a performance reduction of the intended architecture. In Android Apps, it could be more notorious, due to the limited resources of a mobile device, very different from a desktop device or server-type device.
- Software quality decrease: When architectural changes are made in a software project, it could affect the normal behavior of the application and, inside its source code, could imply bad code features implementation, decreasing the software quality, an important standard in a software project.
- Software Sustainability: The cost in terms of human resources, software infrastructure, time, money, etc, could increase if you do not attend to the architectural erosion insights in your software project, affecting non-functional requirements, and, in the long run, affecting the user performance.

2.3.4. Metrics and treatments for architectural erosion

Today. Some different metrics can determine architectural erosion in different approaches. According to Baabad et al. [1]. Some metrics have been created to analyze architectural decay in open-source projects, analyzing possible reasons, indicators, and solution strategies for it. In the resume, there are around 54 metrics that could determine architectural erosion in different stages, from the design stage to the deployment stage. Those metrics have been classified by measured artifacts, level of validation, usability, applicability, comparative analysis, and support tools. Different classifications could be implemented into a tool with a specific measure strategy for analyzing architectural erosion in any system.

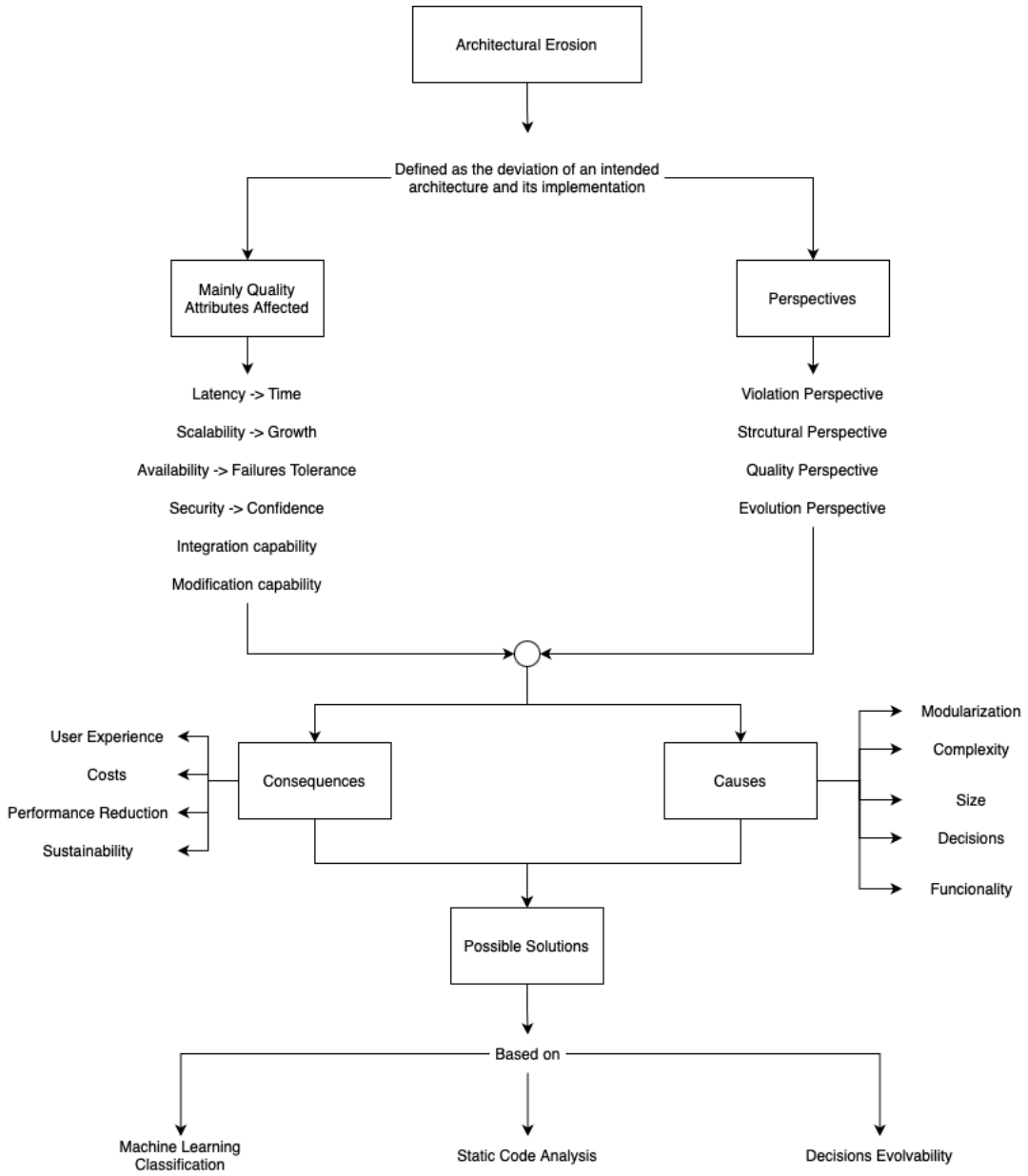

Figure 2.4: Main Concepts of Architectural Erosion in Software Engineering

2.4. Programming Languages fundamental for Software Analysis

With the fundamentals of programming languages, it is possible to analyze the software building. Programming languages have a defined vocabulary, structure, and semantics. Those features make the use of AI tools and NLP techniques more effective. Furthermore, it is possible to create custom rules for checking a defined set of guidelines for a standard structure of different patterns in a software project. This is possible by detecting different semantical, grammatical, and lexical patterns inside the source code of any software project. In general, a programming language is defined by the following components:

- Lexical component: In that component, we define the vocabulary and the set of words that will have a meaning for the programming language. For example, the word function in JavaScript programming language means a function declaration, or int in Java, which means the Integer primitive data type. It is necessary to define all the words that could be used in any source code file of that programming language.

- Grammatical component: With a defined set of words in the lexical component, the next step is to define the order in which words could be written in a code block. It is essential to define all the possible structures that could be defined in any program and the different ways that could be written. For example, most programming languages used in the industry have a defined structure of if statements and all the possible ways to write them.

- Semantical component: If we have a set of words and a defined order to write them, it is possible to build a kind of translator for a programming language. In this step, we define the type of translator with two options: for executable program building, that is, a compiler, one example of it is C language programming, and a semantic visitor that generates an executable program through a translation process between C code fragments and machine instructions. The other option is to make an interpreter, where, in execution time, we visit the line by line of code and translate it into a machine instruction; an example of that approach is the Python programming language. In both approaches, we specify the

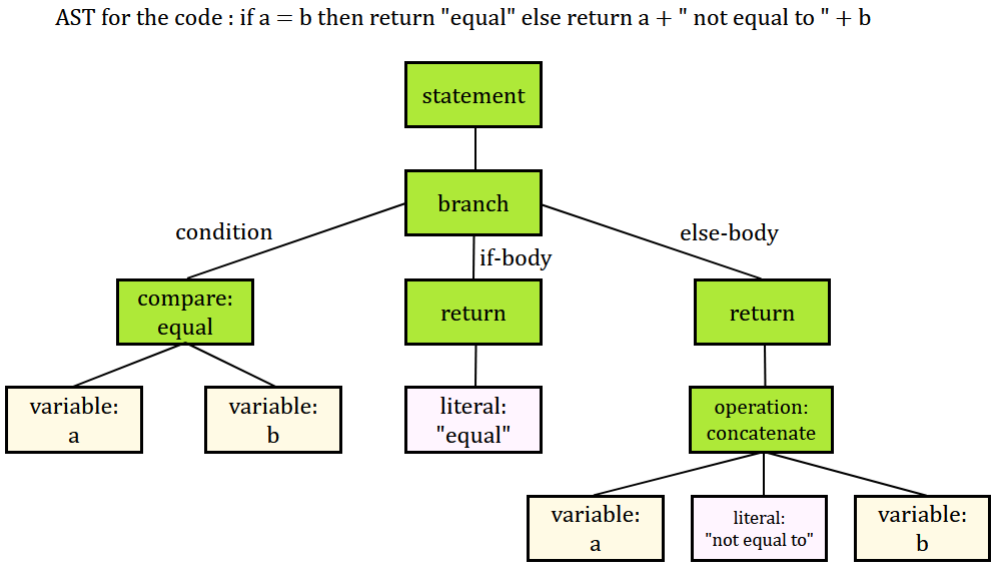

Figure 2.5: Example of an Abstract Syntax Tree (AST) statement.]

translation strategies mainly with two components: a visitor component and a call graph component.

- Visitor component: A visitor component consists of a structure built from grammatical and lexical components of a programming language. In that structure, we can find how all the statement code blocks are defined in all source code files of a software project. We can find the name of every parameter declared in any function and the name of any class of any code statement defined in any source code file. With this component, we can detect any pattern in names and data types inside all code fragments and combine them for more customized check rules (security issues, connectivity issues, and others).
- Call Graph component: The call graph component is very similar to the visitor component. The main difference is that we can find all the dependency relationships between all the source code files of a software project. With this dependency structure, we can detect the high dependency between components and their high coupling rate.

```
private static void Main(string[] args)
{
    if (!args.Any()) return;
    var integer = ParseSomeData(args[0]);
    Console.WriteLine(integer);
}

private static int ParseSomeData(string data)
{
    Console.WriteLine(data);
    return int.Parse(data);
}
```

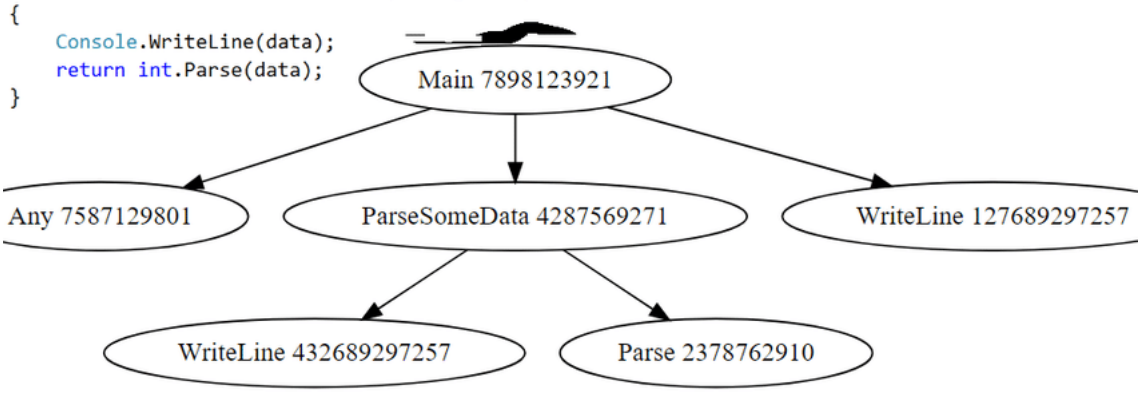

Figure 2.6: Example of an Abstract Syntax Tree (AST) statement.]

With all the programming language components and the way of translating it to machine instructions, it is possible to make custom check rules for different violations detection to different standards. In architectural erosion detection, these concepts will be fundamental.

2.5. Natural Language Processing in Software Issues Detection

Before solving architectural erosion insights into a software project, you have to detect the real violation types that could appear in your software project. There are some approaches for software issue detection using actual natural language processing methodologies that have been powered with Machine Learning Techniques. Natural Language Processing (NLP) gives the ability to extract relevant information from large text sets known as a corpus. Nowadays, NLP is very useful for many tasks, like text generation or classification. Even with a different approach, the NLP actual tools have had a better performance of the same tasks in code due to their standard structure, which does not present different language variations that could be present in a spoken language.

NLP has a set of different preprocessing methods for language models. Due to the complexity

of text representation for computer processing, it is necessary to define a standard input structure for a model language. Before this, you have to make a series of processes for getting the standard structure of a given corpus of text. The process that enables those actions is named text normalization Jurafsky [4]. With the use of regular expressions, you could build a dictionary of words, which is useful for getting a standard structure and enabling the text words for the modeling task. The main stages and processes for the dictionary building are:

- Tokenization: Given a character sequence, in this case, a sequence of words of a given context, you can split that sequence into minimal processing units named tokens. These tokens are normally defined in terms of words. These tokens will create the base dictionary to begin the text processing into a language model Manning [7].

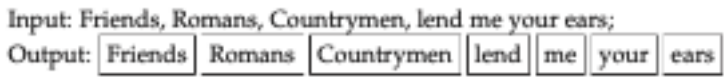

Figure 2.7: Tokenization process of one sentence Manning [7]

- Removing Stop Words: In the basic language modeling tasks, it is necessary to remove words that do not have relevant semantic information inside a text or a text corpus; those words are named Stop Words. Stop Words are words that do not contribute to the meaning of a sentence, like prepositions, articles, etc. In actual language models (Large Language Models), it is very important to maintain Stop Words to get a better specific context for next-word prediction or sentence classification. There are different strategies for removing those words; the most common is removing by collection frequency, due to the number of appearances in the corpus, which is enormous compared with relevant words. In document retrieval, the rare words are the most important for giving an efficient model over the text corpus Manning [7].

- Lemmatization: For new tokens controlling and token derivations, it is essential to create tokens from the roots of the words, to reduce inflectional forms and related forms of words with the same root. In some cases, it is difficult to implement that process because you must have a root word dictionary to get root tokens. In this case, in different contexts,

could generate conflicts for getting roots of specific context words [7].

- Stemming: This process consists of a heuristic process to cut off some characters at the end of each word, reducing the derivation of some words, with the same objective as the lemmatization process. In English, language could be an efficient technique, but it could have some conflicts with other languages.

2.5.1. Word Embeddings for Word Representation

As said in the last section, language models need to have a numerical representation of the corpus text for modeling tasks. To solve this, you must define a standard structure based on the decided model inputs. The most common structure is a word embedding representation, where you define a numerical vector to represent a specific context (where the embedding was trained) for use as input into a language model. This representation gives all model vocabulary a vector representation, where you can observe similar words, different words, and how much distance is between them. This representation is useful for similarity word management and getting the relationships between different features inside them Jurafsky [4].

Figure 2.8: Representation in two dimensions of Similar Words gives a Word Embedding Jurafsky [4]

2.5.2. Performance and Similarity Metrics in NLP

Maintaining and defining an objective in terms of performance is very important. Different metrics represent the behavior of a classic or modern language model based on next-word probabilities

(like the anagram model when you generate an n-tuple of words and calculate the occurrence probability of that word sequence). For information retrieval, when you, in the same case of word embedding representation, have a vector representation, you must use a metric based on the vector's components. In the same dimension, you could determine the similarity between two vectors and verify the semantic similarity between two words in an NLP context. One of the most used metrics in this approach is cosine similarity. This metric combines the product of the two vectors and the difference between their components. The Similitude Cosine metric is defined as:

$$cos(v, w) = \frac{v.w}{|v||w|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} w_i^2}\sqrt{\sum_{i=1}^{N} v_i^2}} \tag{2.1}$$

With this metric, we can conclude the similarity between two words (or two documents without another research context). If the similarity value is high, the words can be considered similar.

2.6. AI models and Software Engineering

With the mentioned NLP fundamentals, different machine learning models have been built from software engineering quality processes, like code refinement, code quality processes. and code generation with better performance compared to the given source code from any software project. All the machine learning models have a basic component that enables their computational processing. This component is a neuron, a mathematical model of a neuron of the human brain. The neuron is the basic processing unit that, through a regression model and an activation function for learning different patterns from the input dataset, depending on the determined learning task for the model. With layers of a series of neurons, it is possible to build a neural network for learning complex patterns from the input data. With that concept, different basic models have been built for different learning tasks:

- Logistic Regression: The logistic Regression model is a machine learning model that acts a the processing unit of a deep neural network. Logistic Regression is a supervised machine learning algorithm for classification tasks based on linear regression and gives a probability that any data belongs to any output category. First, we use linear regression with all the

data features. After that, we use a function like the sigmoid function to get the probability of each data row. The Logistic Regression could vary depending on the output classes (binomial or multinomial).

$$z = w_0 + w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n \tag{2.2}$$

$$\sigma = \frac{1}{1 + e^{-z}} \tag{2.3}$$

- Decision Trees: The Decision Trees model is a model that tries to build a tree structure for predicting the class of any data input. Asking about every feature of every row of data, it classifies depending on the achievement of any feature or not. The main disadvantage of this model is the tendency to overfit and a poor performance with very different data compared to the training data.

- Perceptron model: With the concepts from the Logistic Regression model, the perceptron model is built. The perceptron model acts as a simple human neuron, with the use of linear regression and any activation function like the sigmoid function or the ReLu function, that classifies the input data into the output classes

- Deep Neural Network: With the concept of neuron extracted from the perceptron model, a new model can be extended. The neural network model is the basic Deep Learning model. The neural network model collects a set of neurons that act as the basic processing units and "learn" the main features of the input data in the training phase (validation data is recommended for this kind of model). Deep Neural networks allow for more complex data as images and multimedia files. This kind of model is the starting point for building more complex models and architectures. For learning tasks related to natural language, it needs an attention mechanism is needed for handling the main features based on the context of any text corpus.

Despite the accuracy of those models for some learning tasks. They have any disadvantages for more complex and structured data. With the fundamental concepts, it is possible to build more complex machine learning models according to more complex learning tasks and their data input structure. One of these kinds of models is the models are based on the Transformers architecture.

2.6.1. Transformer Architecture

Some architectures of machine learning models use an attention mechanism to get the main features of a large amount of data in natural language. The first one is the Long Short-Term Memory (LSTM), which holds some features of data from previous learning steps. This kind of model is used for learning tasks of input data that depends on previous data in the learning stage. However, the LSTM model does not hold the features of a large text corpus or many previous steps. Furthermore, the LSTM models demand a high rate of computational resources in terms of memory and computing processing for training the model based on the previous steps or previous data. To optimize that model, in attention is all you need, the Transformer architecture model is proposed [**?**].

The Transformer architecture uses attention heads as an attention mechanism named the attention heads. An attention head uses three matrices that act as three roles: the query matrix, the key matrix, and the value matrix. Every matrix helps to retain all the important information from many previous steps. The Transformer architecture uses a normalization layer and a feed-forward layer for mixing the positional encodings of the input data with the result of the processing of the heads' attention component. Another component is the Word An embedding model that combines a positional encoding to consider the information in the input data. The Transformer architecture is the basic model for building the actual Large Language Models (LLM) and derived models for specified learning tasks, like encoder-only models for classification tasks, decoder-only for generation tasks, and the autoencoder models that combine classification and generation learning tasks. In software engineering, it will be useful for developers' message treatment for bug and code smell detection, and for source code treatment.

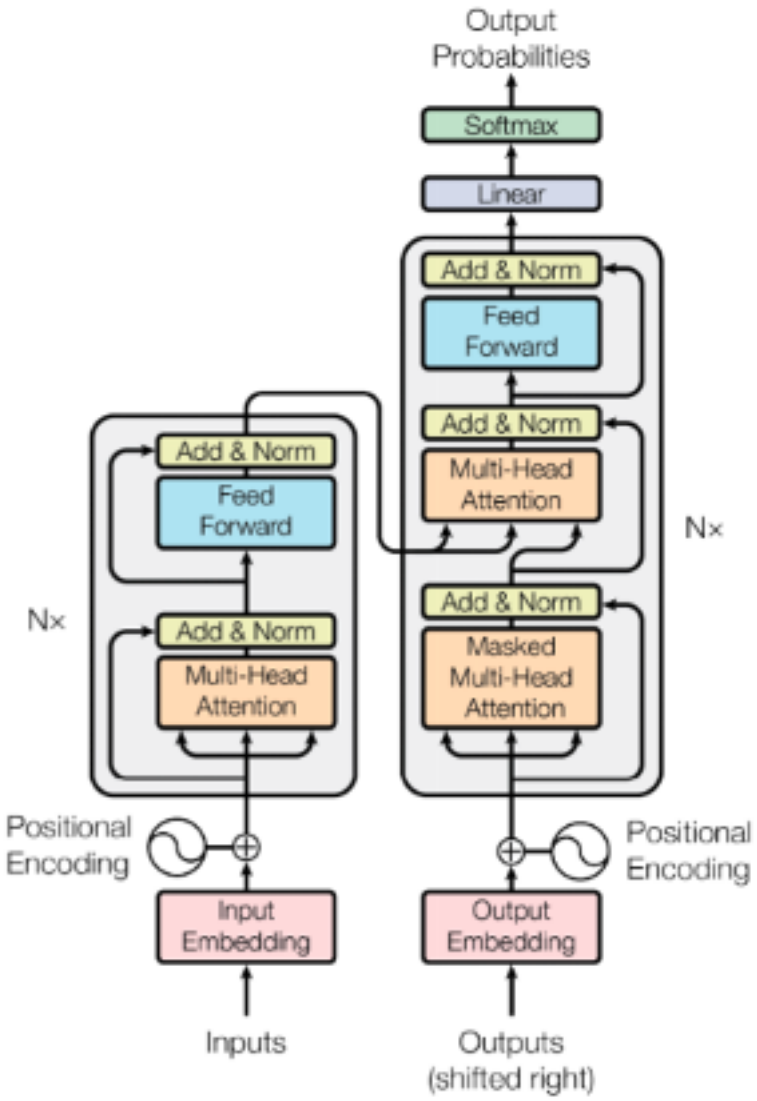

Figure 2.9: the Transformer architecture for machine learning models **?**]

Chapter 3

Related Work

3.1. Research Methodology

For architectural erosion symptoms, causes, and consequences identification, during the research, it was necessary to realize a series of steps for identification and possible recovery processes. Due to the orientation of the actual architectural erosion solvers for solving them, mostly for Backend and Frontend projects, it is necessary to identify architectural erosion symptoms in Android apps. After that, with human judges, these symptoms have to be confirmed, and, finally, those symptoms will generate architectural rules for detecting them and suggest different recovery ways during the coding stage of a system.

3.1.1. Architectural Erosion: An Initial Overview

The first approach is to define and explain the concept of architectural erosion in software engineering, and to set the relationship between this concept and the mobile development ecosystem. In the first approach, we look for advances in static analysis solution approach in architecture quality gates in Server-Side and Frontend applications *reference paper uniandes*. In this case, we locate the cited related work in this research to search for the initial motivation for solving architectural erosion issues and find quality gates in terms of performance in the Android ecosystem.

The first overview we extracted about the related in Ruiyin et al. [9]. In this paper, we find

a Systematic Literature Review (SLR) that defines the definition, reasons, symptoms, consequences, and solution approaches to architectural erosion. In the resume, this approach finds 73 relevant papers about 8 research questions related to the last-mentioned features. From this approach, we can execute the same query in different research papers databases for more current research, due to the publication year of this research (2021), and the time period criteria (between 2006 and 2009) for finding recent research and advances during the last three years. The paper searches relevant papers in 7 research databases and performs a better-performed query, due to the relation between the "architectural erosion" concept in civil engineering and a phenomenon presented in buildings. We present the first set of search queries and their databases:

Query
("software" OR "software system" OR "software engineering") AND ("architecture" OR "architectural structure" OR "structural") AND ("erosion" OR "decay" OR "degradation" OR "deterioration" OR "degeneration")

Table 3.1: Executed Query for related work search

Since those first results, we can extract relevant papers that include new developments and approaches for fixing architectural erosion issues, metrics, tools, and more related work. With this, we identify in each paper three main stages for solving those issues. In the identification stage, we use different identification alternatives through developer messages in versioning systems like GitHub. The identification Stage, where we use Model Driven Development (MDD) and code patterns identification for detecting the identified architectural violation rules. Finally, based on the solution approach (Design approach, quality approach, etc.), we can suggest a proposal for solving the detected architectural violations. In the first search iteration, we added another filter related to the publication year. We selected the papers whose publication year is between 2021 and 2024, which include in their bibliography the first SLR that uses as a reference research. Despite different troubles with the query, when different databases show papers related to civil engineering or architecture (this is because the architectural erosion definition in that context is another research topic about building degradation. In this first research iteration,

we found researches that synthesize different solution approaches and gives an overview from different perspectives, since the process of AER issues identification, metrics that could indicate a possible architectural violation in a software project, like coupling metrics, and the relationship between classes. We observed that the use of NLP techniques could improve the performance of AER issues identification based on commit message analysis. With the use of pre-trained Word Embedding models trained in a specific context, different AER issues are detected, and a list of potential keywords could be generated as a type of alert of an architectural violation based on the architectural model base. In the symptoms and causes approach, an analysis of different base applications made in different programming languages was conducted. Different alternatives and methodologies were evaluated to determine the performance in architectural erosion issues identification, and different solution approaches were proposed.

3.2. Architectural Erosion Symptoms Identification

In the stage of identification, the latest researches give feedback about identified architectural erosion symptoms and their types during different project stages. However, the detected and named symptoms are oriented to Frontend and Backend development. For mobile development oriented to Android technology, specifically made in the Kotlin programming language, it doesn't exist no repository of possible architectural erosion symptoms. The most recent researches use NLP techniques based on GitHub commits. The researchers analyze the main keywords written in GitHub commits that could indicate a bad architectural issue implementation, identifying a possible architectural erosion issue. With a large amount of data from Git repositories is possible to make an NLP analysis of Github commits and define metrics that identify (from previously selected commits tagged by expert judges in software engineering and software architecture) similar keywords for architectural erosion. The metric's performance could be affected by the word embedding training context. The most recent research that implements that identification methodology uses a Word Embedding model trained with 10 million Stack Overflow posts, a context that uses technical definitions for defining features, bugs, recommendations, and more of different programming languages. With this context, word embeddings could be more powerful and efficient for detecting architectural erosion issues, keywords [6, 3]. Another approach is sim-

ilar to the one mentioned previously. However, those approaches are only based on architectural conformance checking. That consists of a set of different statements for a couple of judges who are professionals in software development and software architecture

3.3. Metrics that could indicate Architectural Erosion

Due to recent research, it is possible to determine the difference in metrics between an implemented architecture and an intended architecture, the main reason for the generation of the phenomenon of architectural erosion. In the resume, there are around 60 detected metrics in the software development process and in applications' source code analysis that could affect the maintainability of a software project. Research selected large software projects written in traditional programming languages like Python and Java, and, with a set of judges, selected different source code implementations and past papers that try to describe and identify with similar methodologies. However, those metrics were detected in other development stacks like Frontend development and Backend development, developed with traditional languages and frameworks. However, in the Android development context, the defined metrics could vary according to the mobile software development process for different reasons, like the recommended architectures for Android development and the mobile software development stages. For this reason, with the reviewed data for this research, we need to find different patterns that would indicate an architectural erosion issue and create a relationship between the reliability metrics found in recent research with the architectural erosion issues found in Android applications' source code. In this research, it was employed different papers were employed that found a set of metrics in different analysis code platforms that use different methodologies, mainly static analysis code techniques. The research consists of a Systematic Literature Review (SLR) that found different studies from different research databases. The study found 43 relevant papers for architectural erosion metrics. Founded metrics are defined with different criteria statements like historical data revision, architectural complexity, architectural dependency coupling analysis, or architecture size analysis. These metrics were found from open-source software development projects to industrial software development projects. Different measurement strategies established the effectiveness of the collected metrics. These strategies were found from different code analysis tools like Sonar-

Qube, CKJM, etc. Most of the detected issues are mapped to non-functional requirements inside a software development project, but most of them are related to maintainability, architectural issues, and customized quality gates determined by architectural deterioration or evolution.

Furthermore, there are different metrics and reasons to handle good programming practices with different architectural standards. Based on the main quality attributes in software architecture, it is possible to determine control metrics that could identify improvements with the suppression of identified AER issues in the source code of an Android application. In the availability quality attribute, one of the most common problems during the development stage of a software project is the bad implementation of exception handling. Bad exception handling in a software project could affect the complete flow that achieves a functional requirement and could generate catastrophic software failures and the possibility of a crash of an application. Despite this, some tools use custom lint check rules to detect common bad implementations of exception handling in software development. However, the tools spent considerable machine resources, like computer processing time and RAM. [**?**]

Additionally, the latency and scalability quality attributes have been implemented with asynchronous programming techniques for better performance in the Android operating system. The recommended architecture standards and guidelines give guidance about the inappropriate use of blocking library functions for data or behavior change operations. The use of synchronous programming techniques in any layer based on the MVVM architecture could affect the use of resources of Android devices. For that, it is necessary to implement the use of coroutine techniques and avoid synchronous operations in mobile application development. In the identification of AER issues stage, it will be very important for testing AER rules and detecting blocking functions in Android source implementations [**?**].

3.4. Giving Architectural Erosion Solutions

The latest researches give feedback about identified architectural erosion symptoms and their types during different realization project stages. However, the detected and named symptoms are oriented to Frontend and Backend development. For mobile development oriented to Android technology, specifically made in the Kotlin programming language, it doesn't exist no repository

of possible architectural erosion symptoms. To get this, we use an artificial intelligence approach for detecting architectural changes, and those changes and their change messages (commits in the git world) are useful for identifying possible symptoms and generating rules to prevent them [**?**].

3.4.1. Static Analysis Code techniques

Architectural erosion is a general phenomenon that occurs in all fields of the software development process. Recent research for architectural erosion identification and identification has been focused on Backend development and Frontend development. For this, different static analysis tools have been created as plugins of different Integrated Development Environments. IDEs. One example of that is the use of Antlr-determined grammars in the Eclipse plugin for bad pattern identification in (Data Transaction Object) DTO files. Those files on Backed development are used for service exposition and communication with other components inside a software project. There are similar components that use static analysis code techniques in the industry. The most common platforms are SonarQube and different linters with custom lint rules.

With static analysis code techniques, different advantages exist for architectural erosion issue identification. One of them is the ease of detecting patterns in terms of dependency classes, coupling components, class name standards, and package name standards. Furthermore, we can extend that identification approach to detect customized architectural rules for a specific software development process. [**?**]

However, in the Android software development context, no platform considers the architectural erosion identification and identification process with architectural erosion metrics or customized quality gates. For this reason, it is necessary to define an identification process to detect different parents in different components. The pattern identification process must be based on a specific standard or a specific recommended architectural pattern for Android application development. With the main concepts of programming languages, it is possible to define custom lint check rules with different tools like linters and IDE plugins. Static analysis code technique could help mobile developers to find architectural violations to define non-functional requirements in the development stage.

3.4.2. NLP techniques and AI Models

For the architectural erosion identification process (and identification, but mainly the AER identification process), different tools could be useful for architectural violation identification through Natural Language processing fundamentals. As an additional feature to that field, it is possible to use different AI models powered by different training and contextualization techniques to get a better performance in the AER issues identification process. Furthermore, the models present different alternatives to generate corrected code according to the detected issue in the AI training stage.

In research by Li [**?**], the identification methodology is based on the use of pre-trained Word Embedding models. Word Embedding models are essential for building machine learning models based on the transformer architecture. Before that, word embedding models were implemented in different models. According to the two types of word embedding (static and dynamic Word Embedding models), it is useful for specific use cases. This research implements known models based on static Word Embedding models like Glove [**?**], trained by a large corpus of text from different websites.

For the AER identification process, different AI model sets have been implemented for the AER identification process based on developers' messages extracted from a code versioning platform like GitHub, GitLab, or OpenStack. After a large identification process based on human judgments. With this identified AER issues dataset, basic AI models have been proven for potential key identification that could indicate architectural violations.

Basic models like decision trees, multilayer perceptrons, and other classification models like Gaussian and Naive Bayes machine learning models are used for reliability software measurement. With a defined set of metrics that define the coupling rate between components of any software application, or metrics that define the dependency rate, inheritance depth (in the object-oriented programming paradigm), and cohesion rate. With those metrics, the different AI models were trained for code smell identification concerning software reliability quality attributes. The dataset consists of a set of software projects with the mentioned metrics. All the mentioned metrics are numerical, an important aspect for the use of the mentioned basic machine learning models [**?**].

The results of the proposed research showed the correlation of some mentioned metrics, which

could be related to basic implementations in specific instances of the development stage of the software project, and the effectiveness of the use of machine learning models for finding patterns related to whether defined metrics of any software project or its implementation over time in different versioning platforms [**?**].

Furthermore, different AI models with a large number of parameters and based on the transformer architecture were trained with a large amount of commits from different source code versioning platforms. The extracted commits were from different software applications built with programming languages like Java, C++, and Python. Around three million code lines from those projects were extracted for training this model. One of the AI models is CodeBERT. CodeBERT is an autoencoder model for multitasking purposes:

- Generate code with better performance compared to an input code fragment.
- Explain possible errors and issues in a given code. Those explanations are provided in natural language (English in this case due to its training process).
- Give a score based on the code quality of a given source code fragment.

The CodeBERT AI model was trained by more than 3 million lines of code in different programming languages. Based on that model, Microsoft presented a set of models derived from the CodeBERT AI model. The variations with the other models are the learning task, the source code input format, and the architecture of each model. This model could be useful for a training process based on code generation for getting a better performance and code quality concerning AER smells or AER issues. Another model derived from the CodeBERT model is CodeReviewer. The CodeReviewer AI model was trained for three learning tasks: code quality estimation, code refinement, and review comment generation. This model is based on the Roberta transformer model and was trained with millions of source codes of the most popular programming languages. In the future work section, we will talk about the use of that model for code quality based on AER smells and AER bad implementations.

Another machine learning model trained with other classification and generation tasks is the T5 model. This model was developed by Google and was proposed as one of the first machine learning models for code generation. The model is an autoencoder model for getting a code

fragment from a requirement written in natural language and for code generation with better performance compared to a code fragment as data input. A similar model with the generation task is the NL2 model [**?**].

There are other proposed models with more complex architectures and larger amounts of data in terms of source code from software projects and commit messages from different versioning platforms. Those models are tested with different benchmarks and have shown effectiveness for some learning tasks that do not demand a large amount of data (in terms of millions of lines of code). Those mentioned models will be relevant to extend the objective of this research for code classification in terms of different quality attributes to get a better performance in Android apps.

Another useful tool powered by AI and NLP techniques is the use of AI Agents. AI Agents are models that make an easy connection with large language models. AI agents allow request automation to large language models with different quantities of data, optimizing costs and time to resolve different learning problems. This is useful for different development problems like AI chatbots and automated responses generation [**?**].

AI agents, in the case of a code review task, could be useful to make judgments and revisions of code fragments. With a customized prompt and batching techniques with large language models, it is possible to evaluate large amounts of code fragments and measure the effectiveness of different code tagging according to a specific criterion. In the case of AER issues identification, we will use an AI agent to measure the effectiveness of the evaluated AI models

3.4.3. Metrics Analysis to detect AER Issues

Comparing metrics over time could be considered another identification approach to AER issues. Different metrics of a different set of applications have been compared to detect some issues in applications, according to the user reviews and the metrics over time of each application, like the number of releases per month, releases per year. With the help of NLP techniques, it is possible to detect potential keywords related to different issues. This solution approach has been widely adopted to detect accessibility issues for different types of applications. In social media applications, this approach has been used to detect issues and opinions related to the accessibility

of blind people. The same for platforms for different countries, detecting issues related to app internationalization and interface issues. [**?**].

This solution approach could be useful to adapt to AER issues. AER issues identification would be a complex task with user reviews. However, it is possible to explore a combination of metrics and reviews, processed with NLP techniques, and detect possible issues related to the architecture of any application.

Chapter 4

Research Questions and Scope

According to the research problem about AER issues in Android projects, it is possible to define a set of desired results and objectives about how to explore issues related to AER based on the implemented methodologies in other kinds of research. The main idea is to establish the goals of every method. For the definition of this, we define a set of research questions related to the advantages of identifying AER issues in Android projects and the extensions for solving issues related to other quality attributes, specifically in Android projects.

4.1. Research Questions

1. **How can we identify Architectural Erosion in Android apps?** The desired result of answering this question is to identify the standards, strategies, and policies for finding AER issues in software engineering. According to the research, we can identify the most important approaches to identifying AER issues.

2. **What methodologies can we use to detect AER in Android apps?** The main idea is to explore the most well-known methodologies about AER issues in software engineering in general. We will adapt and extract the main features of each methodology to set them to find those issues in Android projects.

3. **How effective are the proposed methodologies?** With the implemented methodologies for finding AER issues in Android projects, we will test with different approaches to

measure the effectiveness of each one. These results will conclude what the best methodology is to identify AER issues in Android projects. Furthermore, it would define future work focused on detecting more issues that could affect an Android application in general. The proposed methodologies would be helpful to get better applications and better development standards, supporting software sustainability.

4.2. Research Scope

With the defined research questions, we can define the desired results and goals of this research. These must be defined within the scope of the research: **The definition of methodologies to identify Architectural Erosion in Android apps**. Based on this, we define additional objectives to support the main scope of the research.

1. Define methodologies based on the well-known methodologies applied in software applications. With these, we can implement for everyone in any Android project to detect and explore issues in the development stage.

2. Propose standards and policies about identifying AER issues in Android apps. Standards could improve the development of plugins and tools for AER issues detection using different approaches.

3. Test and analyze the results of the implementations of the different studied methodologies.

The research questions and objectives can define a strategy to implement the main scope of the research and identify improvements in finding AER issues in Android apps and their associated strategies.

Chapter 5

Identification of Architectural Erosion Issues with Static Analysis Code

We can extend the solution approaches of AER issues identification in software projects based on static code analysis techniques. In this case, we implemented an analysis over a set of GitHub commits extracted from different Android projects. With the set of commits, we can identify different code fragments and implementations that could affect the architecture quality of any Android project. To get a better performance in AER issues identification, we used a set of keywords from used Word Embedding models and implemented some NLP techniques to identify the most related keywords in the commit messages. After that, we identified possible architectural smells and issues with a manual tagging process. Once the manual tagging was executed, we identified those issues and implemented a set of lint rules with the use of Android lint libraries to identify them inside any Android project. We tested that plugin and those rules with some applications to detect bad implementations in terms of architecture and quality attributes of any Android project. The recent research about architectural erosion identification followed the same process of detection of potential issues in natural language in the commit messages from different version platforms. Furthermore, the study used different NLP techniques, like the use of static word embedding and word similarity metrics, for identifying potential words that have considerable semantic meaning in the development history of any application. With this base process, a list of potential keywords that could indicate an implemented bad architectural issue is created. However, the detection and selection process of different keywords was developed with large software Backend and Frontend projects. Those projects were implemented with traditional

programming languages like Python and JavaScript. Those programming languages' documentation and support are greater than those of languages oriented toward mobile development. For this reason, it is necessary to adapt that identification methodology for the identification of potential words that have a semantic meaning inside commit messages inserted in a versioning platform for Android applications based on their source code. For this purpose, we extracted from 50 open-source Android applications and applied scraping techniques for GitHub history commit extraction. We can collect the committed messages and find similarities with the defined keywords in the research mentioned. A new version of the list of keywords will increase the accuracy of future selection and detection processes for Architectural erosion issues in the source code of Android applications.

5.1. Methodology

To solve the proposed research questions, we define a methodology to design, implement, and test a solution approach based on the static code analysis methodology mentioned in the related work. With a set of commits extracted from different open-source Android projects, we implement NLP techniques with a static Word Embedding model. After that, we define a similarity metric for finding potential keywords in the Android development context. We realized some reviews to show the accuracy with the use of the Word Embedding model. Finally, we selected a representative set of GitHub commits from the Android projects to find patterns and identify architectural smells. Once the architectural smells are found, we implement those patterns and rules into an Android Studio linter and test those with specific use cases from real Android applications and a sample Android application. We will test by comparing with other lint tools implemented for the Android Studio IDE.

We will explain each stage of this workflow and show the results of this research compared with other proposed lint tools.

5.1.1. Selection of sample Android apps

Based on the methodology of the mentioned research, we designed a similar process of extraction and selection process for an initial version of the commits analysis. In this case, we explore

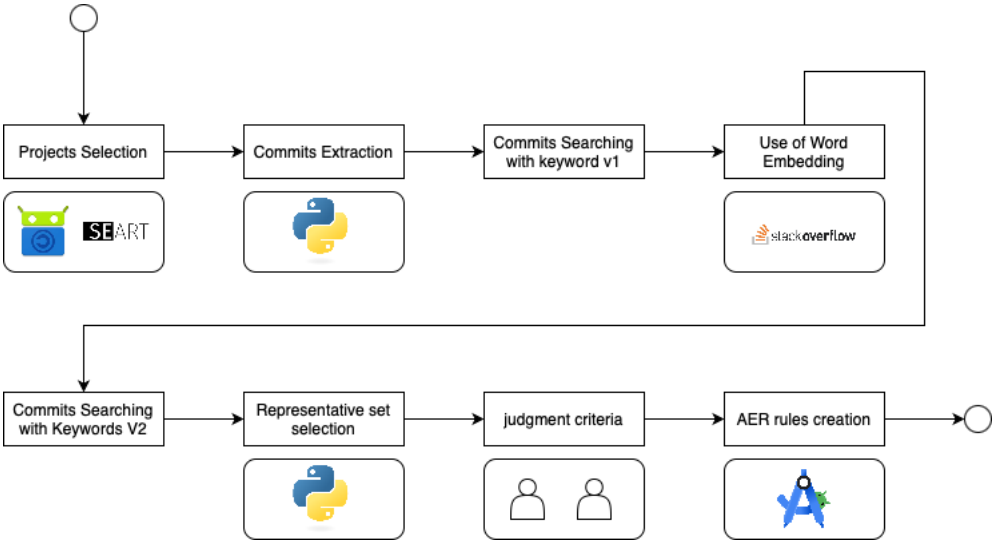

Figure 5.1: seART platform for GitHub repositories searching []

different options and platforms for searching and selecting open-source Android projects whose source code is located on the GitHub versioning platform. We use two tools. The first one is SeART, a platform based on Data Mining repositories research [**?**]. This platform has different filter features that make it possible to customize a search of different GitHub open-source projects. The customized filter for this case is to select GitHub repositories whose main programming language is Kotlin, and the number inside each repository's commit history is in the range of 1000 commits and 30000 commits. This customized filter was selected due to the opportunity to collect a large number of commits for having a strong dataset to identify potential similar keywords with a high similarity metric. With this customized filter, we select the GitHub repositories with the longest number of commits. However, the Data Mining repository techniques used are not the most effective to consider all the GitHub repository environments. For this reason, it is necessary to consider more tools for searching more GitHub repositories of open-source Android applications.

For a larger collection of Android applications for getting a better performance in AER identification and selection, we consider using an Android application searching tool. The mentioned tool is F-Droid- F-Droid is an open-source platform for searching open-source Android applications. This platform gives specific details of every Android application, and one of those details is

the GitHub repository URL. With that attribute and the type of license declared in the F-Droid platform.

Once the applications are selected, we implemented a program based on web scraping techniques with the PyDriller library and built a dataset with the main features of each commit in the GitHub repository of the source code of each Android application.

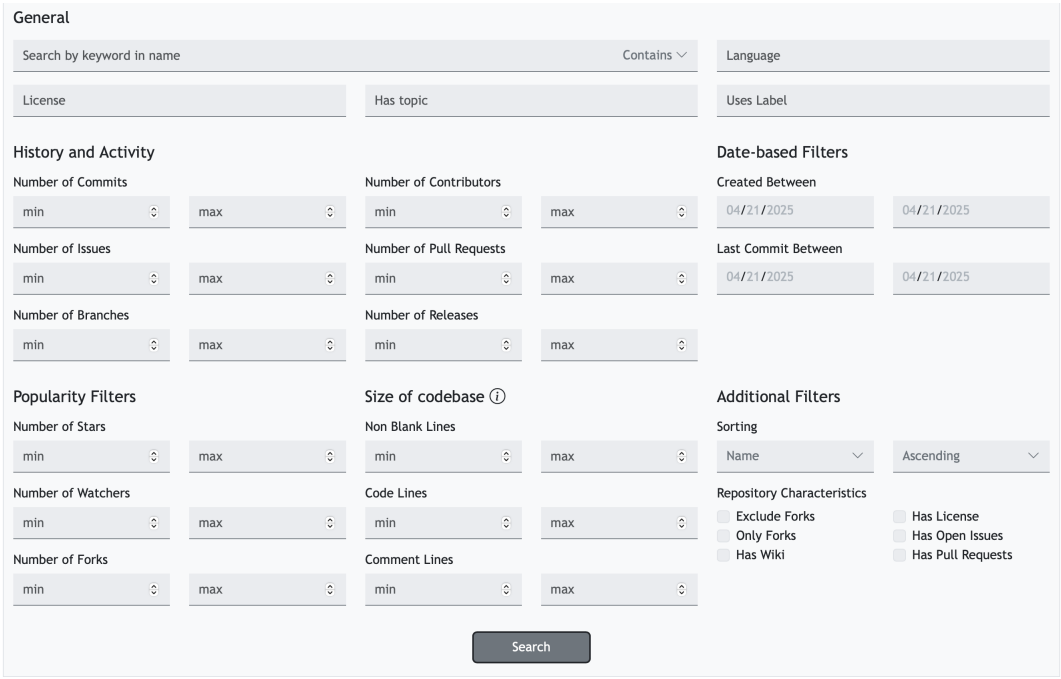

Figure 5.2: seART platform for GitHub repositories searching []

5.1.2. Word Embedding and Similarity Criteria

With the collected set of commits from the mentioned Android apps collection, we made a preprocessing process for the developers' messages of those commits. In this case, we implemented a preprocessing flow with stemming, lemmatizing, and stop word removal. After that, we used the static pre-trained Word Embedding model, trained on 2 million Stack Overflow posts. With that model, we extracted the numerical representation of each word in the vocabulary of the GitHub commits dataset of selected Android applications. With the cosine similarity metric, we extracted the most similar words based on the keywords found in the related work.

5.1.3. Architectural erosion symptoms Identification

Based on the last research, architectural erosion insight identification consists of a deep analysis of information based on development judgments. This judgment can be extracted from different software versioning systems, like OpenStack, a software versioning platform for large-scale software projects, or GitHub, the most used software versioning platform. From those messages, we can classify and tag different code changes through code differences between code changes over time. In this process, the use of different Natural Language Processing (NLP) techniques, to find a standard architectural erosion commit definition. One innovative idea is to employ pre-trained Word Embeddings in software development contexts for potential words that could indicate an implemented architectural violation in server-side applications. The use of embeddings and word similarity metrics like Cosine Similarity allows us to calculate the similarity between the word embedding's numerical representation for every word, since semantic function. With this study, we can standardize the main cases of architectural erosion, the different metrics that could be identified in a software project, and potential solution approaches. With those NLP metrics, we can use them to create discriminatory models for issue detection. However, in an Android context, the architectural erosion identification hasn't been enough to standardize a set of rules for detecting it in software source code due to the Backend and Frontend solution approaches implemented in the study software projects (the two main projects are developed in Python). With the results of the papers extracted in the first related research overview, we can find a set of keywords extracted for the developer's code messages that indicates a potential architectural erosion issue in the implementation, that study was realized with the developer's judges and messages extracted from different version platforms like OpenStack (previously mentioned) and Github, the most popular and used versioning platform. In this case, the project had as a reference four large open-source software projects; all applications are server-side applications. Around 50 keywords were mentioned as potential keywords that indicate an air issue. However, the words were extracted for Backend development purposes. For this reason, we use the same word extraction approach in a mobile context. During this process, we extracted from 50 open-source Android applications published around 470k GitHub commits. This is enough detail to make a preprocessing of vocabulary implemented in each GitHub commit and create new rules

that could be implemented with custom lint check rules. That topic will be treated in the next chapter.

Extracting keywords from Android Context

To get a large set of commits to analyze, we found some platforms with open-source Android projects to extract their code and analyze their commits. We used different platforms based on the Based on the last-mentioned research, it is possible to find potential keywords that indicate an issue or an insight inside a code implementation, with the help of NLP techniques, through similarity measurements like cosine similarity (mentioned in the definitions chapter) and pre-trained Word Embeddings, due to the numerical representation of each word of the generated vocabulary in a specific context. First, we use the PyDriller library *url PyDriller*, a useful library for repository mining, for getting the code source and its attributes of different open-source Android projects made in Kotlin. The selected projects were extracted from different open-source Android project catalogs like F-Droid and other data mining repositories found with different filters like development programming language used, number of commits, and keywords in the selection criteria *url fdroidurl search*. In the first overview, we extracted 50 Android projects that have around 470K commits. With these commits, we made a text pre-processing to build a standard vocabulary and tokenize with the help of the NLTK library *url nltk*, a library for making NLP operations like tokenization, lemmatization, and stemming.

Keyword
architecture, architectural, structure, structural, layer, design, violate, violation, deviate, deviation, inconsistency, inconsistent, consistent, mismatch, diverge, divergence, divergent, deviate, deviation, architecture, layering, layered, designed, violates, violating, violated, diverges, designing, diverged, diverging, deviates, deviated, deviating, inconsistencies, non-consistent, discrepancy, deviations, modular, module, modularity, encapsulation, encapsulate, encapsulating, encapsulated, intend, intends, intended, intent, intents, implemented, implement, implementation, as-planned, as-implemented, blueprint, blueprints, mismatch, mismatched, mismatches, mismatching

Table 5.1: List of initial Keywords extracted of the mentioned related work

Column	Description
Name Repo	Name of the GitHub repository of Android project source code
Url Repo	GitHub URL from source code repository
Commit Message	Message of a specific commit in GitHub commits history of each Android project
Commit Hash	Hash from GitHub commit, essential for the commits analysis process
File Name	List of GitHub commit modified file names
Code Changes	String with the modified source code of each GitHub commit

Table 5.2: Features of commits dataset and their description

With all the corpus from GitHub commits, we implement a text cleaning process, removing stop words and performing stemming, due to the lemmatization process in a technical context is not very effective; some words do not have their respective lexical root, so the tokenizer would not consider those words. The stemming process is useful for semantic word derivation control. This

process is essential because a lot of words do not have semantic relevance in each GitHub commit, so consider that words could affect similarity metrics. With the processed words, we use a pre-trained Word Embedding based on millions of Stack Overflow posts. With the Gensim library *librería de gensim*, we can load the Word Embedding model, get a numerical representation of each selected word, and use the cosine similarity metric for find similar words from the previous keywords. When the metric is generated, we select the 10 most similar words based on that metric. In the first overview implementation, we found around 5000 relevant commits with the updated keywords list. For efficient selection criteria, we extract a representative subset based on a weighted average made by the word frequency in the corpus text. With the first selection of a representative set, we extracted 357 GitHub commits.

Word	Cosine Similitude Average Value
`notion`	0.2674
`respect`	0.2627
`formal`	0.2482
`high-level`	0.2437
`tend`	0.2342
`rigid`	0.2315
`kind`	0.2256
`stronger`	0.2243
`non-linear`	0.2227
`sane`	0.2198

Table 5.3: Top 10 newfound words since Word Embedding cosine similarity metric

With this approach, it is possible to find potential words written in a development context that could indicate a potential issue related to different kinds of functional and non-functional requirements that a software project includes in its architectural design and its standards. This detection approach has many different development areas to detect different problems found in a software project. The future work related to this approach will be discussed in the next chapters.

Chapter 6

Identifying AER With Static Code Analysis techniques

With a set of possible causes presented in Kotlin source code files from various Android projects, it is possible to define a set of rules using the Android Studio IDE tools ecosystem for detecting Architectural Erosion issues. We detected architectural changes in the representative set of commits and extracted different rules and patterns that could handle an architectural violation inside the MVVM architectural pattern for Android Applications. Rules could be implemented in any IDE that supports mobile development oriented to Android, like, for example, Visual Studio Code or Sublime Text (despite its constraints).

6.1. UAST in Android Studio

The Android Studio Integrated Development Environment (IDE) is primarily used for mobile development and building Android applications. Inside its ecosystem, there are a lot of built libraries, frameworks, and other programs that make the development process easier and get better performance in terms of the different non-functional requirements and architectural standards. In addition to those programs, other tools avoid committing any bad practice implementation in terms of different standards based on a specific architectural style. This tool is named linter. A linter is an integrated tool with an IDE for code insight detection and correction (if it is possible). When we write code in Android Studio with bad code practices, its interface shows and marks a possible insight into our code. After that, we select that marker and give a possible solution for

that code insight. Every rule shows its name, its description, a possible reason for the insight, and an optional code implementation solution.

This linter implementation, integrated with the Android Studio IDE, is due to the UAST structure generated by that tool. The Unified Abstract Syntax Tree (UAST) is a defined structure generated by each software project opened by the IDE that contains the AST and the Call graph structures of the source code files of that project. With this structure, we can create different semantical components (similar to the visitors components mentioned in the definitions section) that detect any pattern in different structures and code fragments inside the source code files. We can instantiate the different structures based on the grammar of the Kotlin programming language since there are clauses for classes and others. Furthermore, we can access the different attributes of each structure, like name, source code file name, or package name, and their relationships with other components. In terms of rules, we can define any customized lint check rule, and with that structure's set of Kotlin programming language and its attributes, it is possible to detect any pattern and offer any solution suggestion.

Based on the standard rules implemented in the Android Studio IDE, we can similarly define custom lint check rules for error detection, drawing on the insights reviewed from the representative set of commits in open-source Android projects.

Inside the UAST structure in Android Studio, we can easily implement visitors of the abstract syntax tree. With that representation of the structure of the source code of an Android project, we can instantiate each one of them to analyze it in terms of patterns in its name, its structure, its declaration, and other features that could indicate an architectural erosion issue implemented in its structure.

6.2. Rules Definition

With the representative set extracted from the architectural erosion issues identification process, it is possible to analyze code fragments where the two judges confirmed that it could be a possible architectural violation of the original GitHub source code repository. With that, GitHub commits set and the Google architectural guidelines based on the MVVM architectural pattern], it is possible to find troubles in architectural changes. With these architectural erosion issues, we

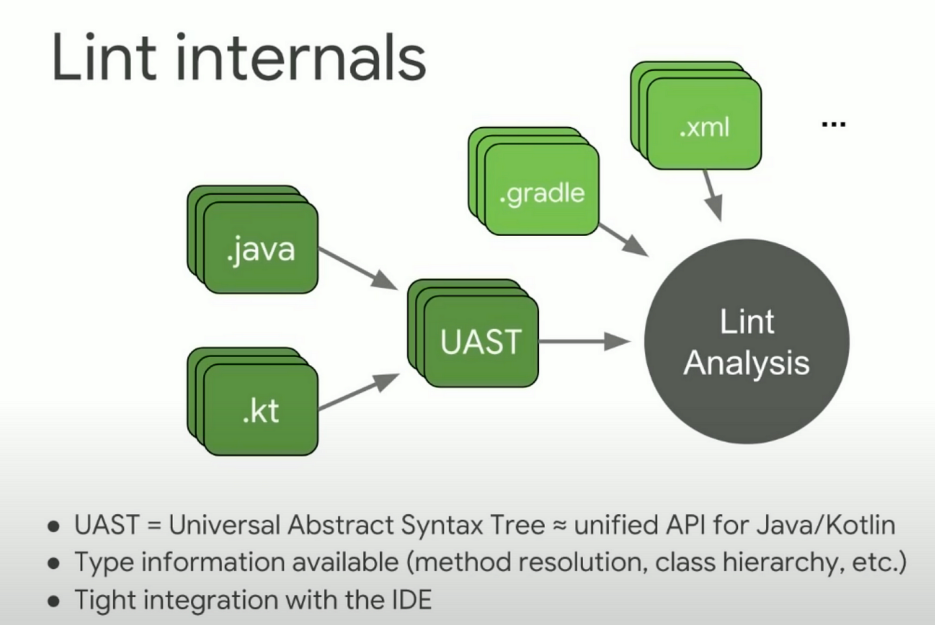

Figure 6.1: Definition of Android Studio UAST structure]

can infer different patterns in terms of class naming, method invocation, dependency injection, and other features that could mainly affect the application performance and other significant architectural requirements inside an Android project. In the table $table_rules$, we present the different found rule sets implemented in the Kotlin programming language, whose implementation could indicate an architectural erosion issue inside an Android project.

ID	Rule	Scope	Description
1	Error Handling Issue	Warning	This rule consists of bad error handling in try-catch statements. Statements like only logging methods or stacktrace println are considered as bad implementations
2.	Blocking Operations Use Issue	Warning	There are some blocking methods of the main thread of an Android application. Functions like blockingGet could affect the normal performance of the main thread of an Android application
3	View Model Error Handler Issue	Warning	Similar to rule 1. It detects bad error handling in stream clauses from live data in the viewmodel layer, based on the MVVM architectural style
4	Bad HTTP client Implementation Issue	Warning	Detection of bad HTTP client implementations. Incomplete configuration that affects latency and the application performance
5	High Coupling Rate Issue	Warning	This rule generates an alert when it finds the most coupled class with another class and components
6	Class Declaration in View Model Scope Function Issue	Warning	No use of component instances, only new class declarations in view model function scopes.

Table 6.1: Features of commits dataset and their description

6.3. AER Detection Component Implementation

To implement the mentioned architectural erosion detection rules for Android applications implemented in the Kotlin programming language, we need to use the lint API toolset given by the Android Studio IDE. It is necessary to understand the tool and how we can implement its functionalities inside any Android project, in terms of version compatibility of different implemented libraries versions, and automatic dependencies injection tool versions, like the Gradle plugin.

Firstly, we need to create an empty Android project. After that, we create a module that

will be the main component for the custom lint check rules. We inject the main libraries to implement the linter's functionality into the project, in this case, the lint API libraries. When the project base has been implemented, we need to declare each one as an issue for the linter. Each architectural erosion rule must be implemented by stating an issue object, where we define the architectural erosion rule issue and its main attributes like name, description, category (in terms of static analysis tools like lint rules, scope rules, and other ones) severity (if the rule could indicate a severity issue, a warning issue, or another category). When we declare the issue object for each rule, we need to add a visitor component that acts as a detector component to each one. To create a visitor component, we need to inject the structures that we should analyze in each pattern of each architectural erosion rule. As an example, for a bad implementation of error handling in different components and layers of an Android application, we need to analyze the try-catch structures. With UAST libraries from Android Studio, we can inject the UTryCatchStatement, which instantiates the try-catch structures found in the Android source code repository. Furthermore, we can add different filters, of which try-catch statements, which we can select by name and code block structure filters.

With the base components of each architectural erosion rule, we can create a custom lint registry template for setting the custom lint check rules inside any Android project. In this file, we create the architectural erosion rules issues based on the previously created issues. After that, we implement that component as an Android library. By this, we need to inject the .jar generated file into the Android application source code; this is possible by making the .jar libraries accepted by the application in the Gradle plugin configuration file. After that, we need to use an XML file, setting the specific name of every custom lint check rule.

With the architectural erosion rules, component creation, and the Gradle and environment parameter configuration. We execute the analysis by calling the functionality of Maven named lint. With this console command $maven --lintargs$, we execute the lint rules since the default implemented lint check rules and the custom lint check rules are declared in the AER component.

6.4. AER Detection Component Testing Criteria

Once we have the AER component with all the implemented architectural erosion rules and the parameters configuration for compatibility of each Android application, we select specific criteria for testing the implemented custom lint check rules based on the analyzed open source applications and their commit set. With the hash commit attribute, we can select a specific instant of that implementation inside the Android project source code repository and observe the custom lint check rules' effectiveness. Due to the different versions of all library ecosystems in Android development, we need to configure the test apps to set as environment variables the Java version compatibility and the Gradle plugin compatibility of the AER plugin. With these observations, we need to select specific test apps and set the environment variables for a correct lint process of each test application. Furthermore, we selected another similar tools that act as a linter in the Android Studio IDE. We will compare the effectiveness of the AER issues detection component with:

- SonarLint: SonarLint is the SonarQube plugin for the Android Studio IDE. SonarQube is a platform based ins static code analysis. This platform detects code smells, security issues, and connectivity issues (in some specific cases). SonarQube is one of the most used code analysis platforms in the software development industry [**?**].

- Detekt: Detekt is a default linter for Android apps and the Kotlin programming language. This plugin detects some code smells for specific Kotlin files or packages of an Android app. This plugin is one of the most used for mobile development, in the code analysis stage [**?**].

- Android Studio default linter: The Android Studio IDE contains an integrated code inspection tool. In that tool, we can define and specify the standard lint rules to analyze inside an Android app. The tool contains connectivity issues, security vulnerabilities, and code smells related to the Kotlin and Java programming languages.

6.4.1. Test Apps Selection

With the detection process previously made, we define a set of three applications where different AER issues were detected with the linter model. The selected applications are:

- AER anti-pattern application: Once the AER issues were identified, we classified specific AER issues per quality attribute category. With this, we consider building an Android application that includes the identified architectural smells. This is to observe the detection accuracy of the AER detection component. As a secondary objective, we will show the performance in terms of AER issues detection of the AER detection component compared to other lint components for the Android Studio IDE.

- Loudius: Loudies is one of the applications analyzed in the AER issues identification stage. Loudious is an open-source Android application consisting of a sample application with architectural guidelines in each layer based on Google's recommended architecture. Loudius is shown as an example of OAuth verification implementation, Jetpack Compose implementation in the UI layer, and the integration with some external components.

- Loudius (different commit): The last selected Android app is the same as the second one. However, this Loudius version was selected from a different GitHub commit. In that commit, we identified some AER issues. This application was selected to measure the effectiveness of the linter component and the other lint applications for AER issues detection.

Those applications were selected by their component structure for observation of the lint detection behavior and the ease of compatibility with the Java JDK version in which the AER detection component was built. With another analyzed application, they do not have the complete injected components available. In addition, some applications are very difficult to implement Gradle configurations for compatibility with the AER issues detection component.

6.5. Detectec AER Insights

Chapter 7

Identifying AER with NLP and AI techniques

Another studied methodology for finding AER issues is related to the use of AI models powered by NLP techniques. We will evaluate the implementation of different static and dynamic word embedding models to extract and define a set of keywords present in GitHub commits of a set of Android projects. With the set of keywords used in the related work section, we will extract more keywords found in a large set of GitHub commits of different Android projects. We measured the cosine similarity average and the cosine similarity between each keyword to define which is the best model to define new keywords that could indicate an AER issue in any GitHub commit [**?**]. Additionally, we will train an AI model to find new keywords and make a comparison with the static word embedding models. This model is CodeReviewer, which is based on the transformer architecture. With this model, it is possible to find more similar words due to the number of dimensions that implement its tokenizer component. Finally, we define a testing stage with manual classification of a representative set of commits for the case of the words generated by static word embedding models. For dynamic word embedding models, we will implement an AI agent with the OpenAI library to evaluate the effectiveness of the trained CodeReviewer model. The main objective of this is to explore and evaluate this methodology for AER issues identification in Android projects.

7.1. Methodology Definition

With the proposed tools and approaches for the implementation of this methodology, we will define a complete workflow to extract and build the commits dataset. After that, we processed the corpus of the commit with NLP fundamentals to find the most similar keywords based on the found keywords that could indicate an AER issue in a code fragment of any versioning platform commit. We used the cosine similarity metric to find the most similar keywords. Those keywords were added to the initial set to find new commits in the extracted commits dataset. Finally, we implemented manual tagging and the use of an AI Agent with a customized prompt to make the comparison of the results between the implemented models: static and dynamic word embedding models.

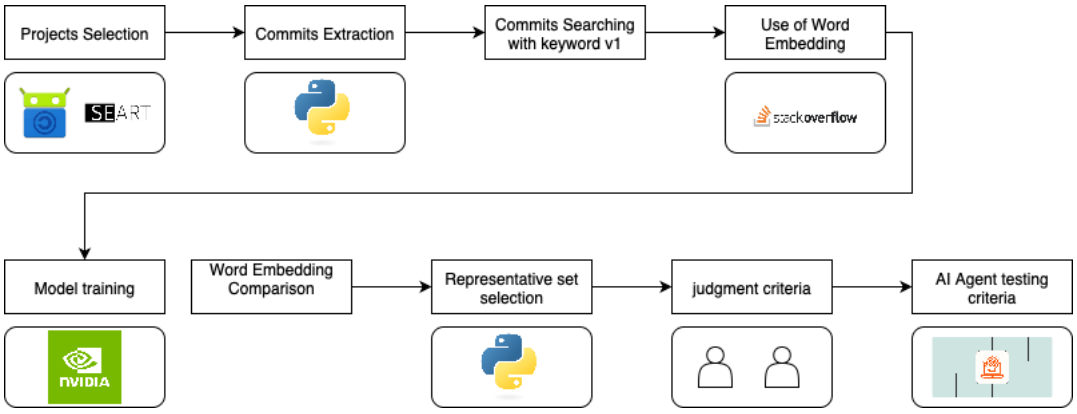

Figure 7.1: seART platform for GitHub repositories searching []

7.1.1. Commits Extraction

The first process is the extraction of a set of commits to evaluate and identify AER issues. We implemented the same methodology for the static code analysis solution approach. We used the same sampled Android projects and used their large set of commits to explore new keywords based on different Word Embedding models.

Applications selections

We used two platforms to build the Android application set: GHS from SEART and FDroid [**?** **?**]. GHS is a repository mining platform of GitHub repositories developed by SEART. We implemented different filters to get some Android applications to the AER issues evaluation. The filters were based on the number of commits (between 1000 and more than 20000 commits), the main programming language as Kotlin, and the date of the last commit. These filters were implemented to gather sufficient information for analyzing the presented AER issues in code fragments written in the Kotlin programming language. The second used platform was FDroid. FDroid is an open-source marketplace with Android applications. FDroid provides some important information about every Android project, like GitHub repository URL and the software license. We randomly extracted s subset of Android applications to add to the previous set of applications extracted from GHS. Summarizing, we extracted a set of 50 Android applications to analyze their commits and perform NLP processing to identify new potential keywords related to the AER phenomenon.

There are some practices and techniques for repository mining. Repository mining allows the finding of relevant information from any source code repository in any versioning platform like GitHub, GitLab, BitBucket, etc. Some tools and libraries have been developed for this purpose. The selected library for this is the PyDriller library [**?**]. PyDriller can extract relevant information from each commit, like project name, commit hash, modified source code, etc. We define a common structure to build the commits dataset. This is shown in Table xxxx. With the defined structure, we implement a Python program to extract from the GitHub repositories of each Android project. The execution of the program generated a dataset of more than 470000 GitHub commits. The commits were filtered if each one contained or not one of the keywords defined in the previous steps.

7.1.2. Use of Word Embedding Models

With the generated dataset of commits. It is possible to explore some alternatives based on NLP techniques. One of them is the use of Word Embedding models. These models help to get a numerical representation of the words in specific or dynamic contexts. Numerical representation

of words helps in computer processing and metrics building. With the two different types of Word Embedding models, we explored the definition of the most similar keywords from the previous set. In the workflow, we defined the use of two approaches to identify new AER keywords: using static and dynamic Word Embedding models.

Static Word Embedding Models

Static Word Embedding models are limited by their training context. We considered the well-known Word Embedding models for text processing, and one Word Embedding trained with a large set of Stack Overflow posts, named the SO model. We implemented NLP techniques like lemmatization and stop word removal, and used three static Word Embedding models: Glove, Word2Vec, and the SO model. We calculate the cosine similarity average to get the top 10 most similar words of each Word Embedding model. We realized the comparison between each Word Embedding model according to the average cosine similarity. Another approach used was to calculate the top 10 most similar words for each keyword of the previous set. However, a large set of words was selected, and some of them were not related to the technological context and the main objective of the methodology. We defined an embedding dimension of 200, which is a standard model for the three analyzed Word Embedding models. Furthermore, we considered a sufficient dimension to represent the words in a specific context.

Dynamic Word Embedding Models

Due to the limited resources of static Word Embedding models, it is necessary to explore additional alternatives to get an improvement with respect to the context of those models. For this, we considered the use of AI models that are trained for code classification and generation. Models like CodeBERT and T5 are small and effective models to classify and generate code based on different training strategies, like code masking, to predict the right keyword in any code fragment [**?** **?**]. CodeBERT is a model that is considered an autoencoder. It can receive different inputs and can generate different outputs for different learning tasks. CodeBERT has distilled models like CodeReviewer, CodeGraph, etc. Distilled models are able to resolve individual learning tasks. In the case of CodeReviewer, its main learning task is to classify code as code that needs review or not. Another learning task in CodeReviewer is the review generation

of code in natural language and code refinement. Based on the review code task, we can extend and implement fine-tuning of the model according to the set of commits of the Android projects, and evaluate the model for AER issues detection according to the commit messages and the code changes of every commit [**?** **?**].

7.1.3. Testing the models

We analyzed the results obtained from the use of static and dynamic Word Embedding models. We will present the results of the extracted words of every model and the performance metrics in the case of the CodeReviewer fine-tuning process. We will make a comparison between the use of the two types of Word Embedding models for identifying AER issues. Furthermore, we will define the advantages and the disadvantages of the use of AI models and NLP techniques for AER issues identification. We will define the possible extension of the use of this methodology for issues identification for other quality attributes in software engineering

7.2. Methodology Implementation

7.2.1. Commits Extraction

We used the commit extraction process outlined in the methodology, based on static code analysis. We used the selected Android applications extracted from the mentioned platforms [**?** **?**]. After that, we used the PyDriller library to extract the commits from the GitHub repository of each Android project [**?**]. As an additional step, we implemented tagging based on the commit message of each row. If the commit contains one of the initial set of selected keywords, the commit will be tagged as 1. Otherwise, the commit will be tagged as 0. This is for an initial tagging for the fine-tuning process with the CodeReviewer model.

7.2.2. Word Embedding Models Results

We implemented each type of word embedding model. For static word embedding models, we extracted the top 10 words and the most similar words of each model to make a comparison of the effectiveness of each model. In the case of the dynamic word embedding model, we extracted

the effectiveness with a sample extracted from the large set of commits of the Android projects. Furthermore, we extracted the top 10 most similar words, based on the cosine similarity metric, averaged with respect to the initial set of keywords. Those results will be tested with an AI agent implementation with the GPT4-o LLM model [**?**]. The AI agent tagged a sample of evaluated GitHub commits used for the CodeReviewer fine-tuning process for measuring the performance obtained with the CodeReviewer model in AER issues identification

Static Word Embedding Models

We presented the most similar words based on the cosine similarity metric, averaged with the initial set of keywords. Additionally, we presented the most similar keywords found by every word embedding model. The top 10 most similar words were shown in the methodology based on static code analysis.

Word	general	respect	design	notion	common	complex	formal	depend	adopt	differ	hing	to-do	borderless	christian	rend	list	misc	non-act	contributor	wiki	conform	disregard	mismatch	implement	distinct	contradict	non-standard	rigid
SO model	0.31	0.3	0.29	0.28	0.266	0.265	0.264	0.262	0.262	0.26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Glove model	0	0	0	0	0	0	0	0	0	0	0.1019	0.1017	0.1	0.09	0.086	0.085	0.083	0.081	0.0809	0.802	0	0	0	0	0	0	0	0
Word2Vec model	0	0	0.2379	0	0	0	0	0	0	0.2269	0	0	0	0	0	0	0	0	0	0	0.2382	0.231	0.23	0.2298	0.22292	0.2291	0.2280	0.2263

Table 7.1: Cosine similarity between Static Word Embedding models

The model with the most extracted words related to the technological context is the SO model. The other models, like Word2Vec and the Glove model, show more words in "natural language", due to the context used for their training. In the case of static word embedding models, it is very important to define the training context and the dimension. The SO model can be considered an important model to explore more issues and warnings in commit messages.

Dynamic Word Embedding Models

For the implementation of dynamic word embedding models, we used an Nvidia GPU with 16GB of memory. We used CUDA for GPU processing for machine learning tasks and AI models management [**?**]. We loaded the CodeReviewer model with the PyTorch library [**?**] and selected a sample of the commits set of the Android projects. The commits selected were around

148K commits, with 50% of the commits tagged as 0 (no containing any keyword of the initial set of keywords), and the other ones were tagged as 1 (containing one or more keywords of the initial set of keywords). We divided the data into training data, validation data, and test data. 70% of commits were used as training data, 15% used as validation data, and the other 15% used as testing data. We implemented the fine-tuning model of CodeReviewer with 3 epochs. We extracted the model performance with respect to the testing data and the tagging criteria (considered as a first approach to select any commit as an AER issue).

Class	Precision	Recall	F1-Score
0	93%	94%	93%
1	94%	93%	93%

Table 7.2: metrics of Codereviewer with respect to test data in AER issues classification

When the fine-tuning process finished. We extracted the tokenizer and the dynamic word embedding model that contains the CodeReviewer model. In the same case as the evaluation of static word embedding models, we selected the most similar words according to the cosine similarity metric averaged with the initial set of keywords.

With these results, we can define a performance metric based on the evaluation of an AI Agent. We implemented with the OpenAI library an agent that used the GPT4-o LLM. We implemented some calls to evaluate a sample of commits evaluated in the CodeReviewer model with a customized prompt that gave the AI agent a specific context.

Snippet 7.1: Defined prompt for AI evaluation

```
You are an expert in software architecture for Android applications
    written in the Kotlin programming language.

You will receive a code fragment extracted from a GitHub commit in an
    Android app repository. This code fragment contains only the lines
    that were changed during the commit:

- Lines starting with ‘+‘ represent **added code**.
```

Word	Cosine Similarity
layout	0.4696
concept	0.4535
package	0.4523
controller	0.4483
settings	0.4440
generic	0.4414
interface	0.4414
element	0.4393
application	0.4281
purpose	0.4280

Table 7.3: Keywords found in the word embedding model of the trained CodeReviewer model

```
- Lines starting with `-` represent **removed code**.

Your task is to analyze **only the lines that were added or removed**,
    and determine whether the changes introduce or contribute to **
    architectural erosion**.

Architectural erosion refers to the progressive degradation of a system's
     design and structure due to poor development practices. It can be
    caused by:

- Violations of SOLID principles
- High coupling between modules or components
- Dispersion of business logic
- Repetitive or duplicated changes across multiple architectural layers
- Leaking logic between layers (e.g., UI handling persistence directly)
- Adding code without tests or separation of concerns
- Implementation of blocking functions in asynchronous contexts
```

```
- Poor or missing exception handling
- Improper exception handling within ViewModel classes

Return only a number:
- 1 if the **added or removed code** could introduce architectural
    erosion
- 0 if not

Code to analyze:
```kotlin
code
```
```

We randomly selected 200 commits for evaluation, constrained by cost limits that restricted the data volume. Commits were chosen based on their length in lines of code. For each commit, we compared the AI-generated labels with those produced by CodeReviewer. The AI agent achieved an accuracy of 55%, which represents a promising initial result for this type of learning task. Further improvements could be achieved by increasing the dataset size and refining both the CodeReviewer annotations and the use of smaller, specialized AI models.

Chapter 8

Metrics Analysis for AER issues identification

Chapter 9

Methodology Extensions

The explored solution approaches for AER issues identification could be extended to identify more issues related to different quality attributes in Android projects. As the developments mentioned in the related work, it is possible to use these methodologies to identify more issues in different layers of the application. The building of plugins and customized linter rules can be extended to identify different patterns to detect different issues in terms of security, availability, scalability, etc. In the same way, it is possible to use different AI models and NLP techniques to solve and detect issues according to the commit message. Nowadays, it is possible to generate a commit message according to the well-written commit standards. With those tools, we can explore the solution methodology approach to detect more issues and potential keywords related to a specific problem. We explored potential research fields whose proposed solution methodologies for AER issues identification could be extended and implemented.

9.1. Static Code Analysis Methodology Extensions

In the case of the use of static code analysis techniques, it is possible to create customized lint rules not only for a general standard of architecture or a specific architectural design pattern. Companies and tools (not only IDEs) can develop customized lint check rules for specific standards and patterns that the company of any specific situation needs

9.2. Use of AI and NLP for Android Applications Issues Identification

AI and NLP have extended the solution approaches in software engineering on a large scale.

9.2.1. Design problems identified by NLP techniques and AI models

Chapter A

TITLE

Bibliography

The references are sorted alphabetically by first author.

[1] A. Baabad, H. Zulzalil, S. Hassan, and S Baharom. Characterizing the architectural erosion metrics: A systematic mapping study. *IEEE*, 2022.

[2] L. Bass, P. Clements, and R Kazman. *Software Architecture in Practice, 4th Edition*. O'reilly, 2021.

[3] V. Efstathiou, Chatzilenas C., and D Spinellis. Word embeddings for the software engineering domain. *IEEE*, 2018.

[4] J Jurafsky, D. Martin. *Speech ad Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. Stanford Web, 2024.

[5] M Laplante, P. Kassab. *Requirements Engineering for Software and Systems. 4th Edition*. O'reilly, 2022.

[6] P. Liang, P. Avgeriou, and L Ruiyin. Warnings: Violation symptoms indicating architecture erosion. *Cornell University*, 2023.

[7] P. Schutze H. Manning, C. Raghavan. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[8] Alexander L Perry, Dewayne. Wolf. Foundations for the stufy of sotware architecture. *ACM SIGSOFT*, 1992.

[9] L. Ruiyin, L. Peng, P. Avgeriou, and M Soliman. Understanding software architecture erosion: A systematic mapping study. *Wiley*, 2021.

Acronyms