The architectural erosion problem in Andorid Apps

Identifying Architectural Erosion in Android Apps

Juan Camilo Acosta Rojas

November 22, 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. Camilo Escobar-Velásquez (Advisor)	Universidad de los Andes, Colombia
Prof. Mario Linares-Vásquez	Universidad de los Andes, Colombia
Prof. Kelly Garces	Universidad de los Andes, Colombia

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 development, teams often carry out a thorough design process in which they define the system components and the resources required for the application. This process is known as *software architecture definition*. Afterward, teams need to align the development process with the established architectural rules in order to meet their goals in performance, security, availability, and other quality attributes. However, due to constant pressure to deliver new releases and frequent feature requests, developers often deviate from the software architecture definition, negatively impacting these system quality attributes. Such deviations can lead to issues related to security, availability, performance, and latency. This phenomenon is referred to as *architectural erosion*. In mobile development, architectural erosion can reduce application performance and degrade other system quality attributes, affecting device resources and ultimately harming the user experience.

Previous efforts have focused on functional and non-functional requirements, such as security, intermittent connectivity, and code smells. However, to the best of our knowledge, no prior work has focused on analyzing the deviation of the development process of Android applications from their defined architecture. This study aims to evaluate the effectiveness of two proposed methodologies for identifying and locating architectural erosion issues.

1.1. Motivation

Previous research has shown that the costs associated with human resources, technology, time, and budget tend to increase after the first release and deployment of a software project Bass et al. [8]. Therefore, it is crucial to identify the factors and artifacts that influence software sustainability and performance in the short, medium, and long term. To this end, various approaches analyze different components of software projects, including their design, source code, and architectural rules. Among these, static code analysis has been one of the most widely adopted and extensively studied Liang et al. [19]. This approach relies on tools that detect and analyze instances of architectural erosion in specific code fragments, often tracking their evolution over time. Although these tools have demonstrated significant value for developers and teams, their application in mobile development remains limited. Mobile ecosystems, such as Android, operate under stricter resource constraints compared to server-side environments, which makes early detection and recovery from architectural erosion even more critical. As a result, there is a need to design specialized mechanisms capable of identifying architectural erosion within Android applications and assisting developers in addressing these issues proactively. Another promising research direction involves applying Natural Language Processing (NLP) techniques to analyze commit messages from version control platforms across Android projects, enabling the detection of patterns related to architectural erosion.

1.2. Machine learning models in architectural erosion

Recent research in Machine Learning (ML), particularly in Natural Language Processing (NLP), has shown that pre-trained models used for word classification or generation tend to perform better with programming languages than with natural languages such as English or Spanish. This advantage stems from the structured and well-defined syntax of programming languages. These insights can be applied to software repository mining to detect issues in source code. In fact, problems such as masked code and design pattern detection have already been addressed using pre-trained ML models.

In the context of architectural erosion, such models could play a crucial role in identifying

early indicators of degradation. Once these symptoms are detected, more specific static code analysis rules can be applied to confirm and address the architectural deviations. In summary, ML models have the potential to serve as an early warning system for architectural erosion in Android projects, leading to optimized components for early detection and recovery.

The objective of this research is to explore two methodologies for identifying Architectural Erosion-Related (AER) issues in Android projects. The first is based on Static Code Analysis techniques, aiming to identify existing work and tools that apply such methods to detect erosion. The second leverages Artificial Intelligence (AI) and NLP models to analyze large collections of commits extracted from GitHub repositories of Android projects. The results of both methodologies will be compared and evaluated to assess their effectiveness and complementarity in detecting AER issues.

Furthermore, this research will examine how these methodologies can be extended to address additional quality attributes in Android applications, such as usability. We propose extending the NLP- and AI-based approach to detect usability-related issues in Android applications, alongside using static code analysis tools tailored for this purpose. Finally, we will explore data-driven analysis of app reviews to identify recurrent usability problems and other emerging quality issues over time.

Chapter 2

Definitions

2.1. Development Process of a Software Project

When creating a software project plan, you must define the core requirements that support the business goals within the software development process. First, you need to specify the functional requirements. In software engineering, functional requirements describe the primary features of each software component involved in the project. Elements such as design, core functionalities, and established workflows are built during the functional requirements definition phase. Next, you must define the requirements that do not directly affect the user experience, known as non-functional requirements. Non-functional requirements indirectly influence the software project; inadequate planning of these requirements can progressively degrade user performance and the architectural structure of the software, resulting in slower, heavier, or less effective applications for individuals. In this phase, you must design a solution that satisfies both the functional and non-functional requirements. This activity constitutes the design phase of a software project. During this stage, you need to determine the set of components that provide the system's functionalities. Additionally, each component's infrastructure must be defined in a process known as Software Architecture Definition Laplante [17]. The software development process consists of several 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, the design of the software components and their relationships can be defined. At this stage, both functional and non-functional requirements are specified. Functional requirements describe the primary features and use cases of the software application, establishing complete interaction flows with the end user. Non-functional requirements, on the other hand, define the performance metrics that the application must meet. According to the ISO standards that define quality assurance for software applications, several categories of non-functional requirements are essential for achieving good performance, including scalability, availability, security, latency, integration capability, and maintainability (these categories will be discussed later). Once the prioritized non-functional requirements have been identified, the rules and standards that will govern the application's performance are established.

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.

2.2. Architectural design in software engineering

For effective and efficient software project development, it is necessary to make the appropriate investment in each stage—from defining functional requirements to designing each component along with its features and constraints. To accomplish this, an architectural design must be created. The architecture of a software project generally defines the components required by the system as well as their connections, relationships, and interaction types. Additionally, each component must be clearly specified within this architectural design. In this way, various standards can be applied to ensure the system's quality in accordance with its defined architectural structure.

2.2.1. Architectural design in Android development

Architectural design in mobile development has progressed gradually, driven by the introduction of new technologies across each development layer—from the data management layer, with

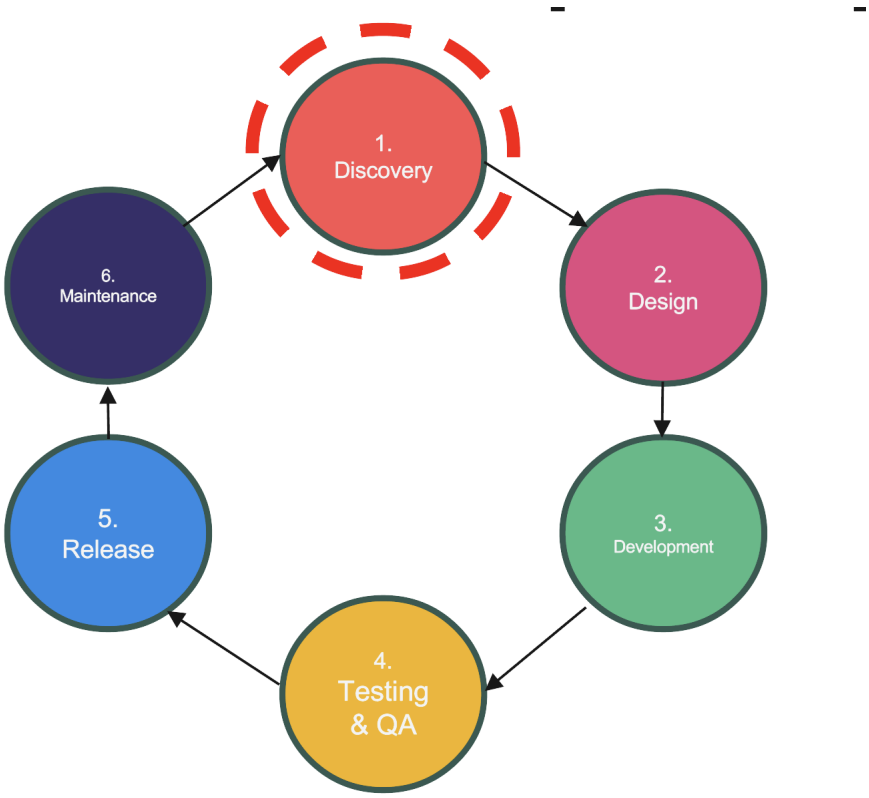

Figure 2.1: The development process of a Software Application

the adoption of new database libraries such as Room, to the user interface layer, where new approaches for building screens have emerged, including Jetpack Compose and Fragment-based interfaces. Depending on the application's complexity, the number of components it includes, which components require interconnection, and the justification for these relationships, it becomes necessary to review the architectural patterns most commonly used in recent years. This review also supports the rationale for selecting one of these patterns as the basis for detecting architectural erosion.

- 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 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,

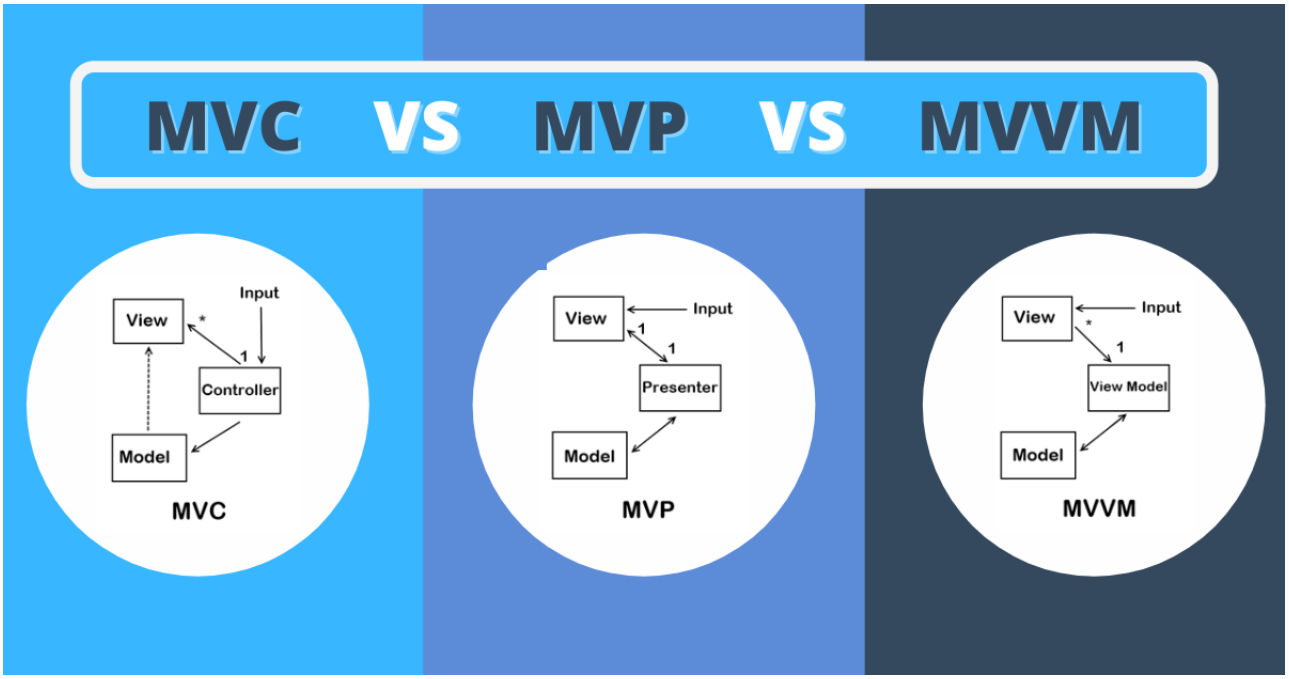

Figure 2.2: Similarities and differences between Android architectural patterns. Gulshan [13]

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. There are six attributes based on ISO-25010 Bass et al. [8]. 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.

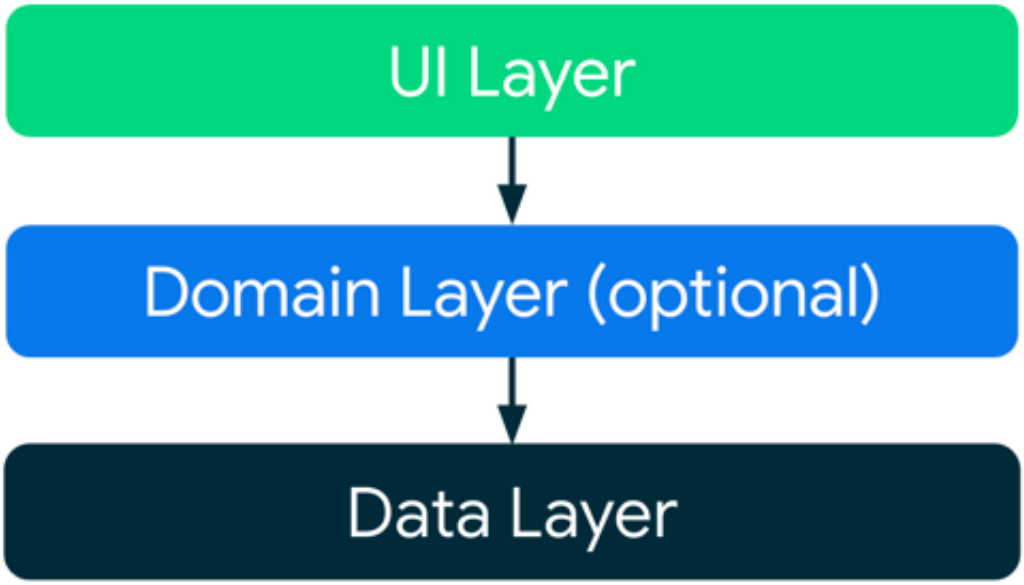

Figure 2.3: Three-layer framework for mobile development. goo [1]

- 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 concept of architectural erosion has been discussed for a long time. Since 1992, it has had a formal definition, establishing its relationship with violations of architectural rules Perry [22]. Architectural erosion can be approached from multiple perspectives, including rule violations, structural degradation, quality deterioration, and evolutionary changes. Additionally, the concept is closely related to similar terms such as "degradation." In a more modern definition, architectural erosion is described as the set of architectural violations that reflect a deviation of the implemented architecture from the intended architecture over time. In summary, in a more concrete form, architectural erosion can be understood as a phenomenon in which the architecture realized in a software project progressively diverges from the one originally planned.

2.3.1. Approaches and Perspectives

Due to the original concept of architectural erosion Ruiyin et al. [23]. 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. Some metrics have been created to analyze architectural decay in open-source projects, analyzing possible reasons, indicators, and solution strategies for it Baabad et al. [7]. 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.

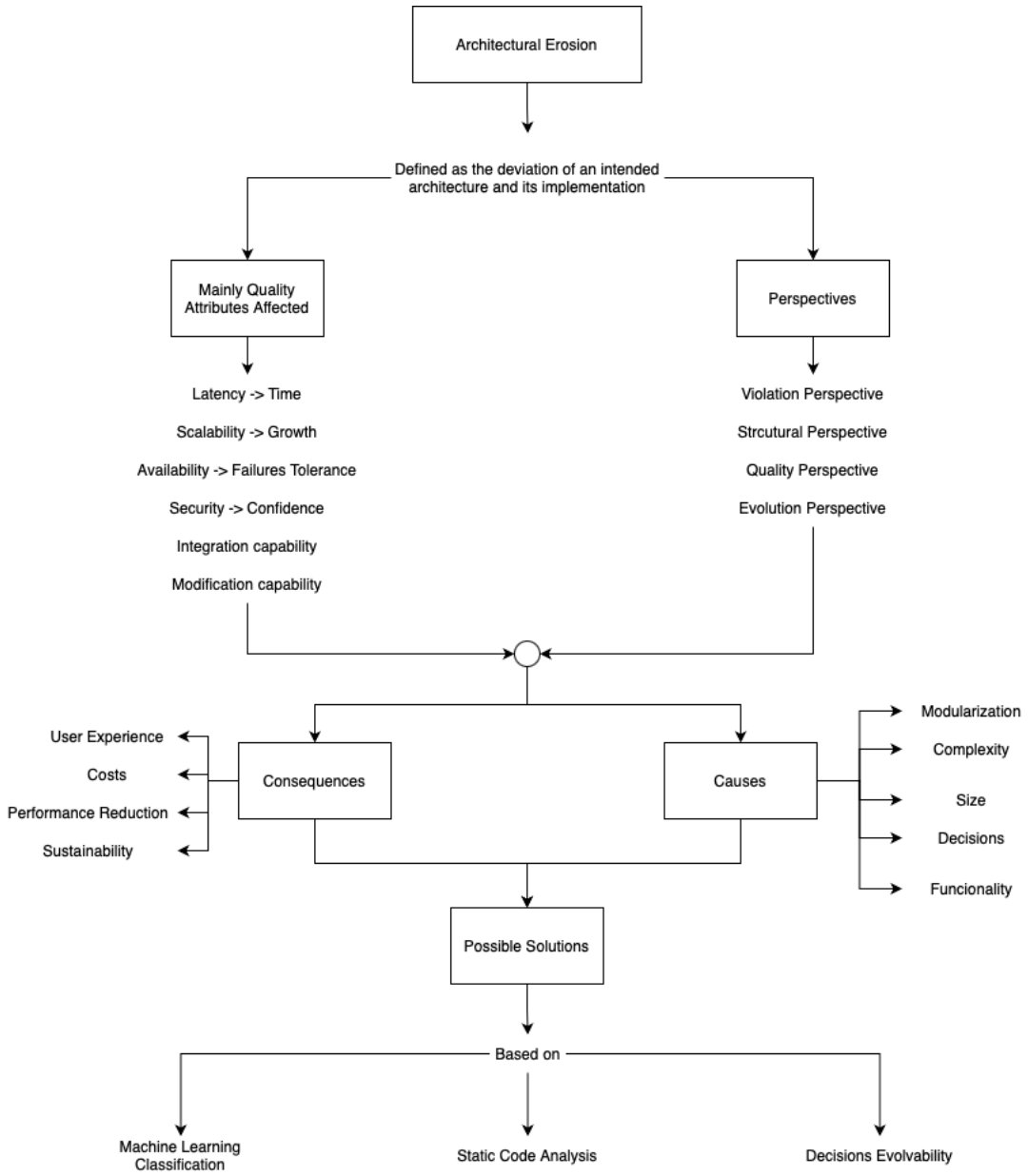

Figure 2.4: Main Concepts of Architectural Erosion in Software Engineering

2.4. Programming Languages fundamental for Software Analysis

With the fundamentals of programming languages, it is possible to analyze software construction more effectively. Programming languages have a defined vocabulary, structure, and semantics. These characteristics make the application of AI tools and NLP techniques more effective. In addition, custom rules can be created to verify compliance with specific guidelines and standardized patterns in a software project. This is achieved by detecting semantic, grammatical, and lexical patterns within the source code of any software system. 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

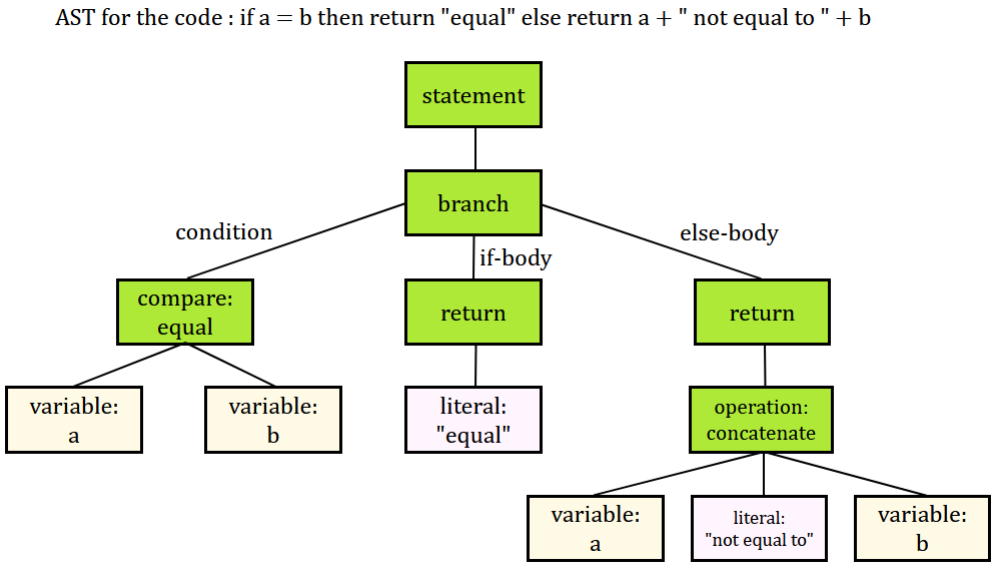

Figure 2.5: Example of an Abstract Syntax Tree (AST) statement. CodeAcademy [9]

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);
}
```

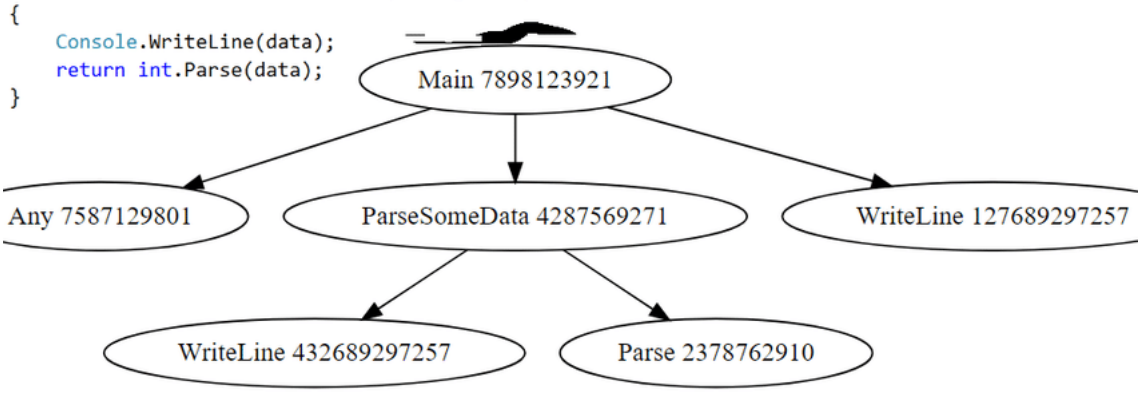

Figure 2.6: Example of a Callgraph of a pair of classes. Staroletov [29]

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 addressing architectural erosion in a software project, it is necessary to identify the types of violations that may occur within the system. Several approaches for software issue detection now rely on modern Natural Language Processing (NLP) methodologies enhanced by Machine Learning techniques. NLP enables the extraction of relevant information from large text collections, known as corpora. Today, NLP is widely used for tasks such as text generation and classification. Moreover, current NLP tools often achieve better performance when applied to source code, due to its standardized and more rigid structure, which avoids the variations commonly found in natural languages. NLP includes various preprocessing methods for language models. Because textual data must be represented in a structured format for computational processing, it is necessary to convert raw text into a standardized input representation suitable for

modeling. This preparation involves a series of steps known as text normalization Jurafsky [16]. Using regular expressions, a dictionary of words can be generated, which supports the creation of a uniform text representation and enables effective processing for modeling tasks. The main stages and processes for dictionary construction include:

- 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 et al. [20].

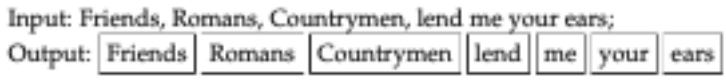

Figure 2.7: Tokenization process of one sentence Manning et al. [20]

- 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 et al. [20].

- 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 [20].

- 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 [16].

Figure 2.8: Representation in two dimensions of Similar Words gives a Word Embedding Jurafsky [16]

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 machine learning model architectures use attention mechanisms to extract the most relevant features from large natural language datasets. One of the earliest approaches is the Long Short-Term Memory (LSTM) network, which retains information from previous learning steps. LSTMs are effective for tasks in which the input depends on earlier data in a sequence. However, LSTMs struggle to preserve information across many steps in long texts and require significant computational resources in terms of memory and processing power to maintain this context during training. To address these limitations, the Transformer architecture was introduced Vaswani et al. [30]. The Transformer uses multiple attention heads as its core mechanism. Each attention head applies three learned matrices that serve distinct roles: a query matrix, a key matrix, and a value matrix. These matrices enable the model to retain and focus on relevant information across long sequences of inputs. The architecture also incorporates normalization layers and feed-forward networks, which combine the positional encodings of the input tokens with the output of the attention computations. Additionally, embedding layers represent words in a high-dimensional space while positional encodings preserve the order of tokens within the sequence. The Transformer architecture forms the foundation of modern Large Language Models (LLMs) and their variants, including encoder-only models for classification tasks, decoder-only models for generation tasks, and autoencoder-based architectures that combine both capabilities. In software engineering, such models can be applied to the analysis of developer messages for detecting bugs and code smells, as well as for the automated processing and understanding of source code.

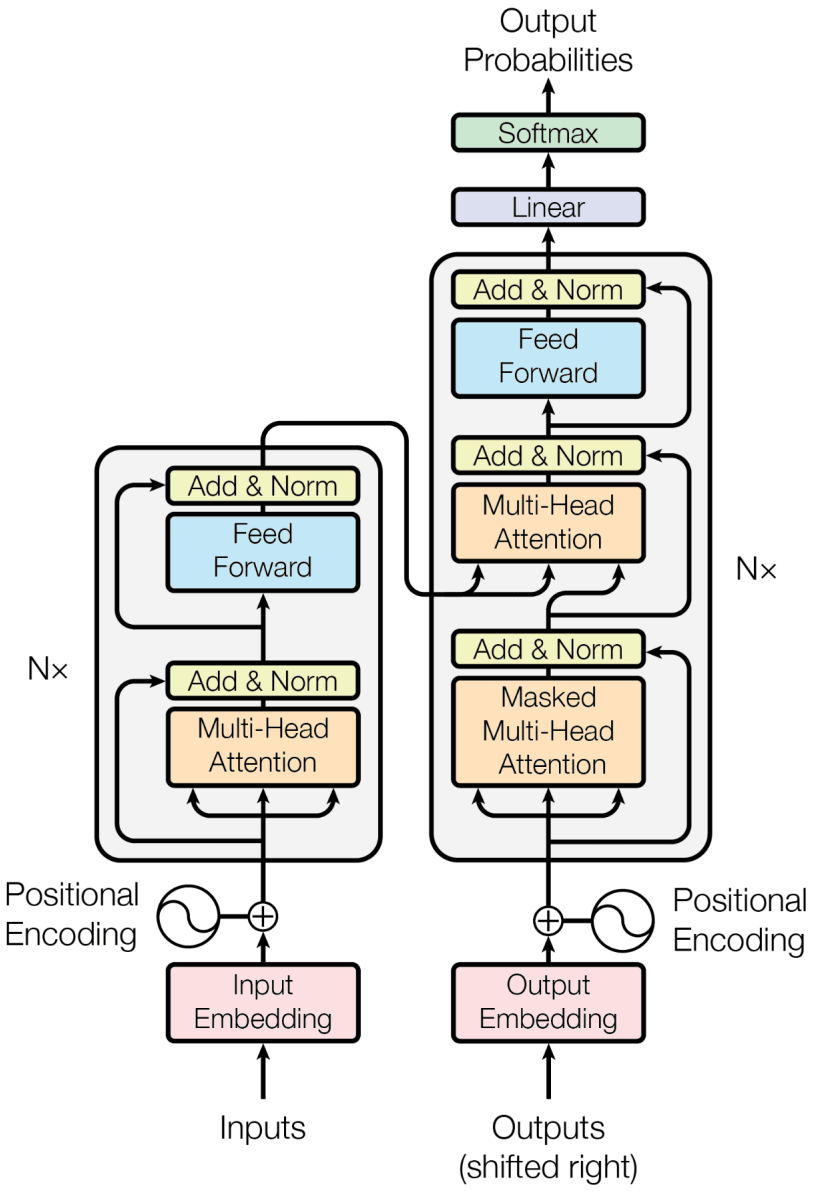

Figure 2.9: the Transformer architecture for machine learning models Vaswani et al. [30]

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. [23]. 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 initial results, we extracted relevant papers describing new developments and approaches for addressing architectural erosion—covering metrics, tools, and related work. From this literature we identified three primary stages for dealing with these issues. First, the identification stage, which exploits developer artefacts such as commit messages in version-control systems (e.g., GitHub) to surface potential symptoms of erosion. Second, the detection stage, which applies techniques such as Model-Driven Development (MDD) and pattern-based code analysis to verify and locate concrete architectural-rule violations in the source code. Finally, the remediation stage, in which, depending on the chosen solution strategy (design changes, quality-improvement measures, etc.), we propose concrete fixes for the detected violations. In our first search iteration we added a temporal filter and retained papers published between 2021 and 2024; these works frequently cite earlier systematic literature reviews as foundational references. During the search we encountered irrelevant results from civil engineering and building-architecture domains—an expected artifact because “architectural erosion” is also used in those fields to de-

note physical degradation. From the selected software-engineering papers we synthesized various solution approaches and produced overviews from multiple perspectives, including processes for AER identification and metrics that may indicate architectural violations (for example, coupling metrics and class-relationship measures). We observed that NLP techniques can improve the identification of AER symptoms when applied to commit-message analysis. Using pre-trained word-embedding models adapted to the software engineering context, it is possible to detect candidate AER issues and generate keyword-based alerts linked to the underlying architectural model. In the symptoms-and-causes line of research, studies that analysed baseline applications implemented in different programming languages evaluated alternative methodologies for detecting architectural erosion and proposed a range of remediation strategies.

3.2. Architectural Erosion Symptoms Identification

Recent research reports identified architectural-erosion symptoms and their types across different project stages. However, these identified symptoms mainly pertain to frontend and backend development. For mobile development—specifically Android apps implemented in Kotlin—no comprehensive repository of possible architectural-erosion symptoms currently exists. Recent studies apply NLP techniques to GitHub commit messages: researchers analyze the keywords found in commits that may indicate poor architectural implementation, thereby surfacing potential architectural-erosion issues. With large volumes of data from Git repositories, it is feasible to apply NLP to GitHub commits and define metrics that—based on a set of commits previously labeled by expert judges in software engineering and software architecture—identify similar keywords associated with architectural erosion. The effectiveness of these metrics can be influenced by the training corpus used for word embeddings. For example, one recent study trained word-embedding models on 10 million Stack Overflow posts, a technical corpus containing descriptions of features, bugs, recommendations, and other programming-related content. In this context, word embeddings may be more effective at detecting architectural-erosion–related keywords Liang et al. [18], Efstathiou et al. [11]. Another related approach relies mainly on architectural conformance checking, which evaluates a set of rules or statements through assessment by a pair of expert judges in software development and software architecture.

3.3. Metrics that could indicate Architectural Erosion

Recent research has made it possible to quantify differences between an implemented architecture and its intended architecture—differences that often lead to architectural erosion. In summary, approximately 60 metrics have been identified throughout the software development process and in source code analysis that may affect the maintainability of a software project. Previous studies examined large-scale projects written in traditional programming languages such as Python and Java, where expert judges analyzed various implementations and compared them with findings reported in earlier studies. However, these metrics were primarily detected in frontend and backend development environments using conventional programming languages and frameworks. In the context of Android development, the applicable metrics may differ due to the particularities of mobile development, such as recommended architectural guidelines and the specific stages of mobile software evolution. Therefore, based on the reviewed literature, it is necessary to identify patterns that indicate architectural erosion within Android applications and establish relationships between reliability metrics found in prior research and erosion issues detected in Android source code. This research draws on multiple studies that identified metrics using diverse source code analysis platforms, most of which rely on static analysis techniques. Conducted as a Systematic Literature Review (SLR), the study collected findings from various research databases and identified 43 relevant papers on architectural erosion metrics. These metrics were categorized according to criteria such as historical revision analysis, architectural complexity, dependency and coupling analysis, and architecture size assessments. The identified metrics were found in both open-source and industrial software projects. Different measurement strategies were used to evaluate the effectiveness of these metrics, employing tools such as SonarQube, CKJM, and others. Most of the detected problems map to non-functional requirements in the software development process, particularly related to maintainability, architectural concerns, and customized quality gates associated with architectural degradation or evolution. Moreover, various additional metrics and guidelines exist to promote good programming practices aligned with architectural standards. Considering the main software architecture quality attributes, it is possible to define control metrics capable of identifying improvements associated with resolving architectural erosion issues in Android applications. For example, within the availability quality

attribute, one of the most common issues during development is improper exception handling. Poorly implemented exception handling can disrupt complete execution flows associated with functional requirements, potentially causing severe system failures or application crashes. Some tools provide custom lint rules for detecting common anti-patterns in exception handling; however, these tools often demand significant computational resources, including processing power and memory St. Amour and Tilevich [28]. Similarly, the latency and scalability attributes in Android development are frequently addressed through asynchronous programming techniques to improve system performance. Recommended architectural guidelines warn against the misuse of blocking operations and synchronous functions within the MVVM architecture, as these may negatively impact device resource consumption. Consequently, the use of coroutines and the avoidance of blocking operations are strongly recommended for Android development. In the architectural erosion identification stage, testing detection rules and identifying blocking calls in Android code will be essential Wang et al. [31].

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 Baabad et al. [6].

3.4.1. Static Analysis Code techniques

Architectural erosion is a widespread phenomenon that affects all areas of the software development lifecycle. Recent research on identifying and managing architectural erosion has been primarily oriented toward Backend and Frontend development. As a result, several static analysis tools have been developed as plugins for different Integrated Development Environments

(IDEs). One example is the use of ANTLR-based grammars in Eclipse plugins to detect poor design patterns in Data Transfer Object (DTO) files, which are commonly used in Backend development for service exposure and communication among software components. Similarly, the software industry offers several platforms that apply static code analysis techniques, such as SonarQube and different linters with customized lint rules. Static code analysis provides several advantages for detecting architectural erosion issues, including the ability to identify patterns related to class dependencies, coupling between components, class naming conventions, and package naming conventions. These capabilities can be extended to detect custom architectural rules tailored to a specific development process or technological environment Ruiyin [24]. However, in the context of Android software development, there are currently no platforms that explicitly incorporate architectural erosion identification using architectural metrics or customized quality gates. Therefore, it is necessary to define a detection process capable of identifying patterns across different components of an Android application. This process must be based on a clearly defined architectural standard or a recommended architectural pattern for Android development. By leveraging core principles of programming languages, it is possible to create custom lint checks through tools such as static analyzers and IDE plugins. Applying static code analysis techniques in the Android development context would enable developers to detect architectural violations during implementation, facilitating the enforcement of non-functional requirements and minimizing architecture degradation at development time.

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.

The identification methodology is based on the use of pre-trained Word Embedding models

Liang et al. [18]. 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 Pennington et al. [21], 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 Juneja et al. [15]. 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 Juneja et al. [15]. 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 Zan et al. [32]. 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 Arora et al. [5]. 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. Sivayoganathan and Ramzan [25]. 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 address the research questions, we establish a methodology to design, implement, and evaluate a solution based on static code analysis, following the approaches described in the related work. Using a collection of commits extracted from multiple open-source Android projects, we apply NLP techniques supported by a static Word Embedding model. We then define a similarity metric to identify potential domain-specific keywords within the Android development context and conduct evaluation steps to assess the accuracy and relevance of the embedding model. Next, we analyze a representative subset of GitHub commits to detect recurring patterns and identify architectural smells. Once these smells are identified, the extracted patterns and rules are integrated into an Android Studio linter and validated through practical scenarios using real Android applications as well as a reference sample project. Finally, the proposed linter is evaluated by comparing its performance against other lint tools available in 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.

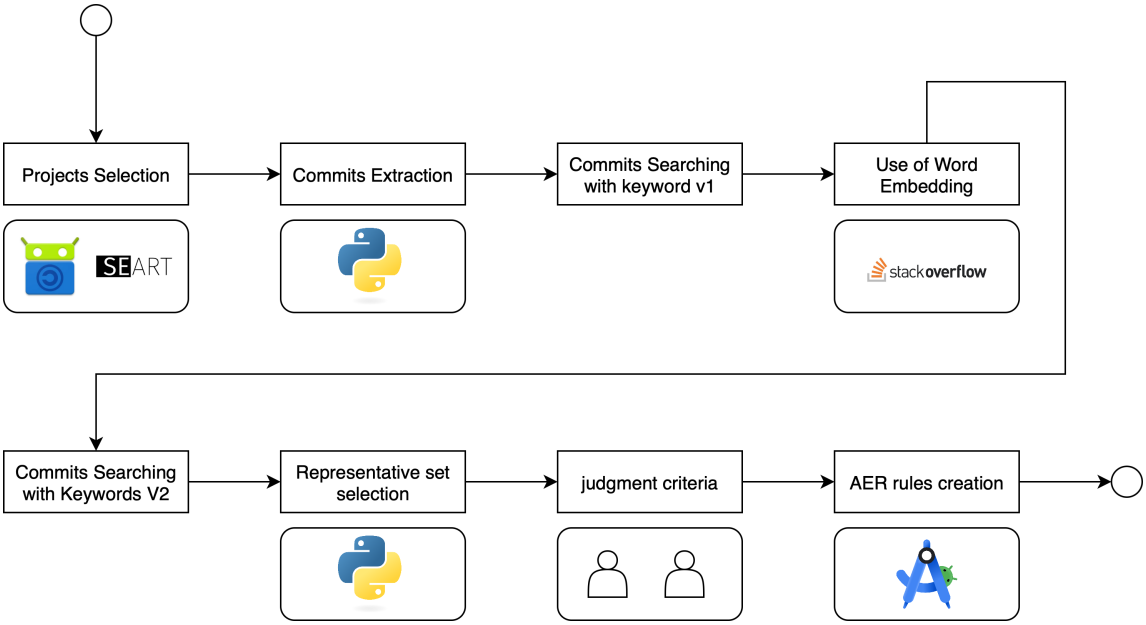

Figure 5.1: Workflow for AER issues identification with Static Code Analysis techniques

5.1.1. Selection of sample Android apps

Following the methodology described in the referenced research, we designed a similar extraction and selection process for the initial phase of commit analysis. First, we explored different alternatives and platforms to search and select open-source Android projects whose source code is hosted on GitHub. For this stage, we employed two tools. The first was SeART, a platform developed for data mining research on software repositories Dabic et al. [10]. This tool provides multiple filtering options that enable customized searches among open-source GitHub projects. In our case, we configured the filter to retrieve repositories whose primary programming language is Kotlin and whose commit history ranges between 1,000 and 30,000 commits. This configuration allowed the collection of a substantial number of commits to build a robust dataset capable of identifying meaningful keyword similarities using similarity metrics. Using this filter, we selected repositories with the largest commit histories. However, data mining techniques alone are not sufficient to exhaustively cover the diversity of GitHub repositories. Therefore, it was necessary to complement the search process with additional tools. To broaden the number of Android applications and improve the performance of AER identification, we incorporated a second tool

focused specifically on Android software discovery: F-Droid. F-Droid is an open-source platform that catalogs open-source Android applications and provides relevant metadata, including the GitHub repository URL and information regarding application licensing. Using this information, we expanded the set of repositories considered for the study. After selecting the applications, we developed a program using web scraping techniques and the PyDriller library to extract and compile a dataset containing key information from each commit belonging to the GitHub repositories of the analyzed Android projects.

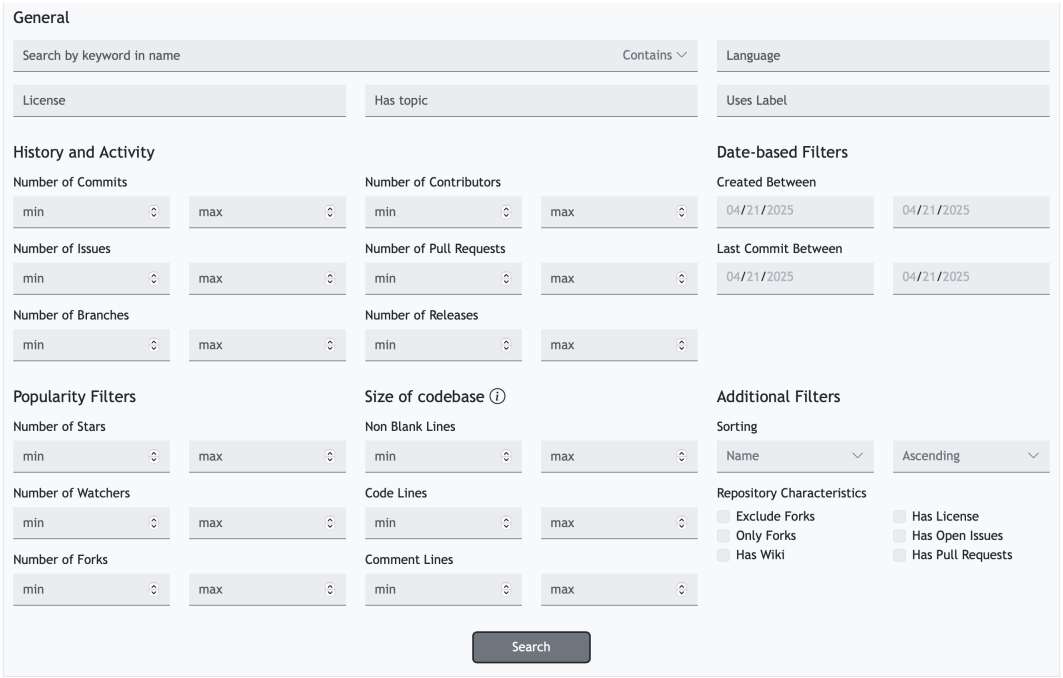

Figure 5.2: seART platform for GitHub repositories searching Dabic et al. [10]

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

According to the findings of the referenced research, the identification of architectural erosion insights involves a detailed examination of information derived from developers' judgments. These judgments can be obtained from various software version control systems, such as OpenStack, a platform used for large-scale software projects, or GitHub, the most widely adopted versioning platform. Using the information in commit messages, it is possible to classify and label code modifications by analyzing differences introduced over time. In this process, several Natural Language Processing (NLP) techniques are applied to derive a consistent definition of an architectural erosion–related commit. A key contribution of prior work is the use of pre-trained Word Embeddings in software development contexts to detect terms that may indicate architectural violations in server-side applications. By applying embedding models and similarity metrics such as Cosine Similarity, it becomes feasible to measure how closely related two words are based on their numerical embedding representations, reflecting their semantic behavior. These methods allow the standardization of common architectural erosion cases, the identification of metrics relevant to software projects, and the exploration of potential solution strategies. With the application of these NLP techniques, it is also possible to design discriminative models for detecting issues. However, in the Android domain, the study of architectural erosion has not yet led to a standardized set of detection rules in source code, largely due to the focus of previous research projects on Backend and Frontend systems—mainly implemented in Python. From the set of studies included in the initial review, a collection of keywords was identified from developer commit messages that could indicate potential architectural erosion issues. This prior study was performed based on developer evaluations and commit histories extracted from version control systems such as OpenStack and GitHub. The research analyzed four large open-source server-side applications and identified approximately fifty keywords as potential indicators of architectural erosion. Nevertheless, these keywords were obtained from Backend development contexts. Therefore, we adopted the same keyword extraction methodology but applied it to

mobile development. For this purpose, we analyzed 50 open-source Android applications and collected approximately 470,000 GitHub commits. This dataset provided sufficient information to preprocess the vocabulary from the commit messages and derive new rules suitable for custom lint detection in mobile development environments. The definition and implementation of these rules will be discussed 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 Spadine [26], 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 fdr [3], Jean de Dieu et al. [14]. 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 Sphinx [27], 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

Using the complete commit corpus extracted from GitHub, we applied a text preprocessing pipeline that included stop-word removal and stemming. Lemmatization was not adopted because, in technical contexts, many terms do not have a valid lemma form, which may prevent the tokenizer from retaining relevant vocabulary. Stemming, by contrast, provides better control

over lexical normalization for technical terms and ensures that semantically related word variants are mapped to a common root. This step is crucial, as many terms within commit messages do not contribute meaningful semantic information and could negatively influence similarity calculations. After preprocessing, we employed a pre-trained Word Embedding model built from millions of Stack Overflow posts. Using the Gensim library GENSIM [12], we loaded the embedding model, generated numerical vector representations for each selected token, and calculated cosine similarity scores to retrieve semantically related terms for the known architectural erosion keyword set. For each reference keyword, we selected the ten most similar terms based on this metric. In the initial implementation, this approach allowed us to identify approximately 5,000 relevant commits using the expanded keyword list. To refine the selection in a more efficient way, we generated a representative subset by applying a weighted frequency analysis of token distribution across the corpus. Using this method, we obtained a final representative set of 357 GitHub commits for further analysis.

Word	Cosine Similarity 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 widely used for mobile development and the creation of Android applications. Within its ecosystem, it provides various built-in libraries, frameworks, and programs that support the development process and improve performance in terms of architectural standards and non-functional requirements. Among these tools are mechanisms designed to prevent the introduction of poor implementation practices according to specific architectural guidelines. One such mechanism is the linter. A linter is an IDE-integrated tool that identifies potential issues in the source code and, when possible, suggests corrective actions. When a developer introduces a code fragment that violates recommended practices, Android Studio highlights the issue within the interface. The developer can

then inspect the warning, which includes the rule name, a description, a justification for the alert, and in some cases, a proposed solution. The linter in Android Studio works by leveraging the Unified Abstract Syntax Tree (UAST) generated for each project. UAST is a structured representation created for the source code opened within the IDE and includes both the Abstract Syntax Tree (AST) and the call graph of the application. With this representation, it is possible to implement semantic components—similar to the visitor structures described in previous definitions—that analyze project files to detect patterns in code fragments. The structure exposes elements defined by the Kotlin programming language grammar, including class declarations and other code constructs, and allows access to metadata such as names, file locations, package identifiers, and component relationships. Using these structures and attributes, developers may define custom lint check rules capable of detecting patterns and suggesting corrective actions. Following the model of the default lint rules included in Android Studio, custom lint checks can be created for detecting architectural violations based on insights obtained from the analyzed GitHub commits of open-source Android projects.

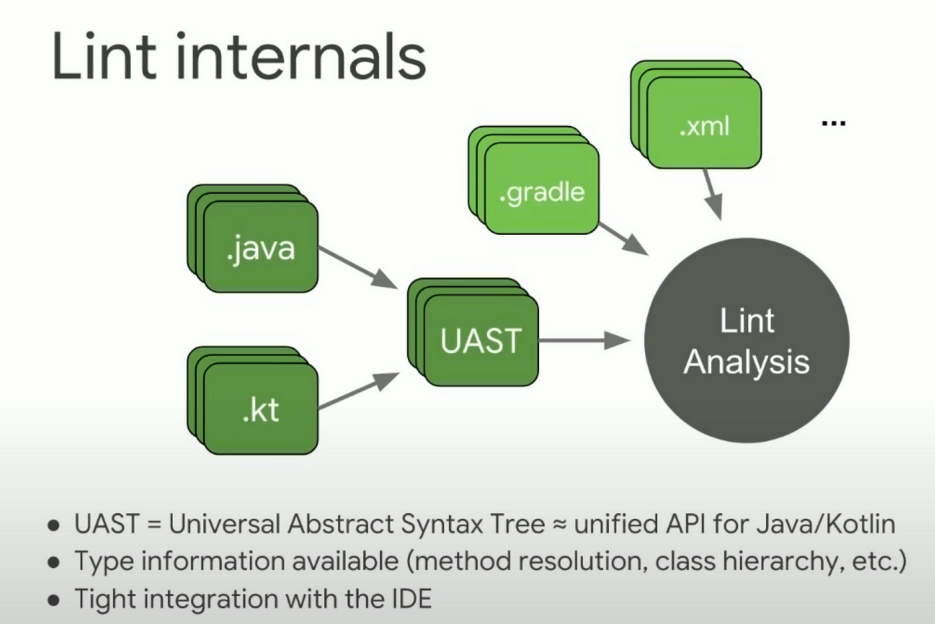

Figure 6.1: Definition of Android Studio UAST structure]

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 goo [1], it is possible to find troubles in architectural changes. With these architectural erosion issues, we 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 architectural-erosion detection rules for Android applications written in Kotlin, we use the lint API provided by the Android Studio IDE and must understand how to integrate its functionality into a project, including version compatibility for libraries and build tools such as the Gradle plugin. First, create an empty Android project and then add a module that will host the custom lint checks. Add the lint API libraries to that module to enable the linter's functionality. For each architectural-erosion rule, declare an Issue object that defines the rule's

attributes (name, description, category, scope, and severity). Each Issue must be associated with a visitor component that acts as the detector; the visitor is configured to inspect the specific AST/UAST constructs relevant to the rule. For example, to detect poor error-handling implementations, the visitor should examine try–catch constructs by leveraging UAST types such as UTryCatchStatement and applying filters on names and code-block structure. After implementing the Issue objects and their visitors, register the rules in a custom LintRegistry and package the implementation as an Android library. The generated JAR can then be included in the application via the Gradle configuration so the custom rules are available to the IDE. Optionally, an XML descriptor can be used to enumerate and name each custom lint rule. Finally, with the AER rules implemented and the build environment configured, run the lint analysis from the console (for example, invoking the lint functionality with the command maven –lint args) to execute both the default and the custom lint checks 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 [4].

- 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 [2].

- 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.

Chapter 7

Identifying AER with NLP and AI techniques

Nota

This chapter was included in a New Idea Paper for A-Mobile 2025 (An extended workshop of International Conference on Automated Software engineering)

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.

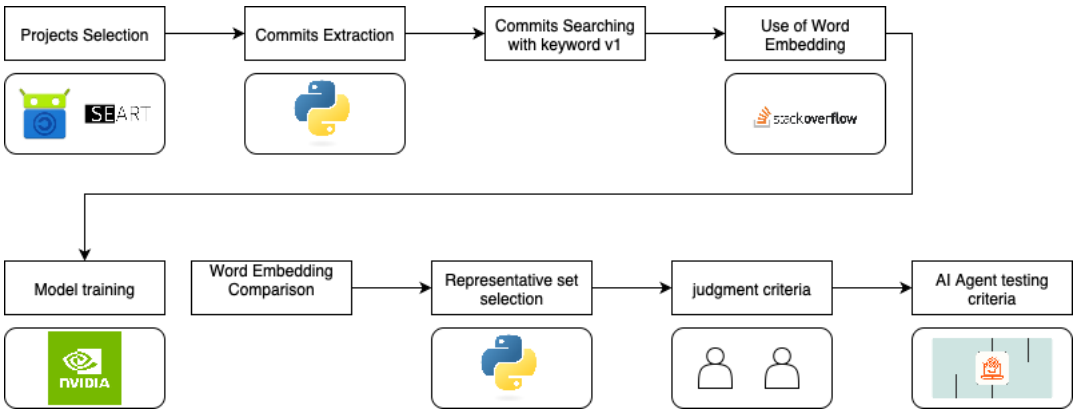

Figure 7.1: seART platform for GitHub repositories searching []

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 ? fdr [3]. 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 Spadine [26]. 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 [**?** 3]. After that, we used the PyDriller library to extract the commits from the GitHub repository of each Android project [26]. 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	ling	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.

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

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**.
- 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

With the studied methodologies for AER issues identification, we can use some patterns to try to identify AER issues related to any context with the help of data analysis and NLP techniques. Another studied approach consists of the use of NLP techniques and scraping of user reviews of the top applications of different categories from the Play Store. The objective of this solution methodology is to analyze a large set of reviews of the most popular applications of every category from the Play Store. With that analysis, we tried to identify potential keywords based on the cosine similarity metric average according to the initial set of keywords extracted from the related work. We collected during 2 months the different features and statistics from the 50 most popular applications of all the categories given by the Play Store, where Android applications can be found. We use some scraping libraries to extract the features of every application in JSON format. After that, we implemented a review scraper with the Google Play Scraper (GPS) library **?**]. From every application, from the initial extraction, we extracted the first 100 reviews and collected them in one dataset. We analyzed the reviews of the generated dataset to find he most similar keywords. With this, we tried to find potential keywords that users use to express any architectural issue or any issues related to some quality attribute, according to the cosine similarity metric. This methodology can be considered as an extension of the proposed methodology to identify AER issues with AI models and NLP techniques.

8.1. Methodology Definition

We defined this methodology based on the solution methodology with NLP and AI models. We implemented two scraping programs. The first one uses the GPS library implemented in the JavaScript programming language. This program utilizes that library to compile the 50 most popular applications in each category. Furthermore, free and paid categories are also considered (See Table 8.1). The second program, based on scraping techniques, uses the GPS library implemented in the Python programming language. This program gets a set of defined reviews of the desired application. We extracted 100 reviews for every application extracted by the first scraping program. With those datasets, we implemented the second identification methodology approach to AER issues identification in reviews of Android applications. We used the SO model to find the most similar words with respect to the initial set of keywords from the related work section [**?**]. Finally, we compared and analysed the results of the most similar words found in all the reviews, and the most similar words found in the reviews of applications of every category.

Category
APPLICATION, ANDROID_WEAR, ART_AND_DESIGN, AUTO_AND_VEHICLES, BEAUTY, BOOKS_AND_REFERENCE, BUSINESS, COMICS, COMMUNICATION, DATING, EDUCATION, ENTERTAINMENT, EVENTS, FINANCE, FOOD_AND_DRINK, HEALTH_AND_FITNESS, HOUSE_AND_HOME, LIBRARIES_AND_DEMO, LIFESTYLE, MAPS_AND_NAVIGATION, MEDICAL, MUSIC_AND_AUDIO, NEWS_AND_MAGAZINES, PARENTING, PERSONALIZATION, PHOTOGRAPHY, PRODUCTIVITY, SHOPPING, SOCIAL, SPORTS, TOOLS, TRAVEL_AND_LOCAL, VIDEO_PLAYERS, WATCH_FACE, WEATHER, GAME, GAME_ACTION, GAME_ADVENTURE, GAME_ARCADE, GAME_BOARD, GAME_CARD, GAME_CASINO, GAME_CASUAL, GAME_EDUCATIONAL, GAME_MUSIC, GAME_PUZZLE, GAME_RACING, GAME_ROLE_PLAYING, GAME_SIMULATION, GAME_SPORTS, GAME_STRATEGY, GAME_TRIVIA, GAME_WORD, FAMILY

Table 8.1: Application categories by the Google Play Scraper library

8.2. Methology Development

We used as a reference the keywords found in the related work section. Those keywords were used to find the most similar terms in a large set of reviews from the 50 most popular applications of every category in the Play Store. We periodically extracted information using the scraper programs, gathering details for each application in the top 50 of every category, including user reviews for each identified application. For two months, we stored the extracted information in JSON files, separating applications into two categories: paid applications and free applications.

For each generated JSON file, we also produced a CSV file to extract and organize the reviews of each listed application. Throughout the extraction process, we collected approximately 720K reviews. We applied foundational NLP techniques to process the large review corpus, including stop word removal and character filtering to clean and normalize the review text. With the processed corpus, we implemented two approaches. For the first approach, we generated a word cloud to identify the most relevant and frequent words in the dataset. A word cloud allows visual exploration of the dominant terms within a text corpus, providing an initial understanding of the general context expressed by users in their reviews.

Figure 8.1: WordCloud of reviews of applications of the Play Store

We replicated the second studied methodology for AER issues identification, powered by AI models and NLP techniques. We used the SO model and got the numerical representation of each word of the processed corpus of reviews. With the set of the numerical representation of each word, we implemented the average cosine similarity metric and found the most similar words related to the initial set of keywords.

Word	Cosine Similarity Average Value
better	0.2227
useful	0.2201
good	0.2070
different	0.1983
really	0.1965
think	0.1948
best	0.1909
many	0.1785
highly	0.1785

Table 8.2: Top 10 newfound words since Word Embedding average cosine similarity metric

The second approach is very similar to the last-mentioned. We separated the reviews according to the applications of every specific category. Categories were extracted from the Google Play Scraper library **?**]. For every separated set of applications and reviews, we implemented the same methodology. We implemented NLP techniques to have a clean corpus of the set of reviews for every category. After that, we extracted the numerical representation of each word and calculated the cosine similarity average. The objective of this approach is to identify potential AER keywords inside specific contexts, like the categories of every set of extracted applications of the Play Store. Results are shown in Table 8.2.

Category	Most Similar Word	Cosine Similitude Avg.
APPLICATION	better	0.2227
ANDROID_WEAR	useful	0.2201
ART_AND_DESIGN	design	0.2883
AUTO_AND_VEHICLES	better	0.2227
BEAUTY	nice	0.1380
BOOKS_AND_REFERENCE	experience	0.1446
BUSINESS	overall	0.2462
COMICS	really	0.1965
COMMUNICATION	important	0.2410
DATING	people	0.1308
EDUCATION	different	0.1983
ENTERTAINMENT	good	0.2070
EVENTS	worth	0.1270
FINANCE	better	0.2227
FOOD_AND_DRINK	deal	0.2037
HEALTH_AND_FITNESS	stronger	0.2243
HOUSE_AND_HOME	easy	0.1274
LIBRARIES_AND_DEMO	useful	0.2201
LIFESTYLE	especially	0.2514
MAPS_AND_NAVIGATION	better	0.2227
MEDICAL	sane	0.2198
MUSIC_AND_AUDIO	really	0.1965
NEWS_AND_MAGAZINES	think	0.1948
PARENTING	kind	0.2256
PERSONALIZATION	poor	0.2157
PHOTOGRAPHY	nice	0.1380
PRODUCTIVITY	useful	0.2201
SHOPPING	better	0.2227
SOCIAL	different	0.1983
SPORTS	stronger	0.2243
TOOLS	good	0.2070
TRAVEL_AND_LOCAL	really	0.1965
VIDEO_PLAYERS	useful	0.2201
WATCH_FACE	better	0.2227
WEATHER	easy	0.1274
GAME	fun	0.2102
FAMILY	people	0.1308

Table 8.3: Most similar word per Google Play category according to cosine similarity.

8.3. Results Analysis and Conclusions

This approach functions as an extension of the proposed methodology for AER issue identification grounded in NLP fundamentals and AI models. The main objective is to illustrate potential research directions for detecting diverse issues in Android applications. User reviews from the Play Store are especially relevant for identifying performance problems that arise in specific execution scenarios of an application. The results obtained from comparing the static Word Embedding model between the initial keyword set and the terms extracted from the review corpus (both globally and separated by category) reveal words with strong semantic value associated with user perceptions of the application. Terms such as *better*, *useful*, and *stronger* appear to reflect user reactions to specific usage cases of the application. Although the detected terms as potential AER-related keywords are not directly linked to a technological context, they can be interpreted as lexical indicators of application performance. These terms may signal potential issues by establishing a connection between user sentiment and application behavior in specific scenarios. Therefore, this study represents an opportunity to detect concerns expressed in reviews that relate to the non-functional requirements of Android applications.

Chapter 9

Methodology Extensions

The explored solution approaches for AER issue identification can be extended to uncover additional problems related to different quality attributes in Android projects. As demonstrated in the related work, these methodologies can be adapted to detect issues across multiple application layers. The development of plugins and customized linter rules can be expanded to identify patterns associated with concerns such as security, availability, scalability, and others. Similarly, various AI models and NLP techniques can be employed to detect and categorize issues based on commit messages. Modern tools even allow automatic generation of commit messages that follow well-established writing standards. By leveraging these capabilities, it becomes possible to further investigate and refine methodologies aimed at identifying additional issues and extracting meaningful keywords related to specific problem types. In this research, we explored potential directions in which the proposed AER identification methodologies can be extended and implemented, demonstrating opportunities for broader detection of software quality concerns in Android applications.

9.1. Static Code Analysis Methodology Extensions

In the case of static code analysis techniques, it is possible to create customized lint rules not only for general architectural standards or specific architectural design patterns. Companies and development tools (not limited to IDEs) can define their own linting rules tailored to organizational standards and patterns required in specific development contexts. Additionally, the

identification of issues can be explored across different stages of development by analyzing components of the programming language. Through the use of semantic pattern detection based on structures such as the Abstract Syntax Tree (AST) and the Call Graph, it is currently possible to evaluate numerous metrics that may signal issues affecting particular quality attributes in software projects. Based on this, for the identification of issues in Android applications, a set of detectable patterns can be defined in terms of syntactic and semantic characteristics of the programming languages used for Android development. From these patterns, customized lint check rules can be created and integrated into the development process. In this way, warnings and recommendations can be generated to support the resolution of multiple issues, such as those related to security, availability, connectivity, and other concerns discussed in the related work.

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. Different issues related to different quality attributes, policies, and standards of software architecture can be identified and possibly solved by the use of AI models. Furthermore, it is possible to use NLP techniques to identify potential words with a high semantical meaning. We identified possible extensions of the implemented methodology for identifying AER issues using AI models and NLP techniques.

9.2.1. Design problems identified by NLP techniques and AI models

One potential extension involves detecting issues within the UI layer of an Android application. Today, various standards and guidelines support the development of sustainable User Interfaces (UI). In terms of application design, two essential aspects contribute to a usable application: accessibility and internationalization. For instance, applications targeting Arabic-speaking regions sometimes experience UI problems, as certain layouts fail to correctly display information in right-to-left text. Similarly, social media platforms such as TikTok have demonstrated accessibility shortcomings, with challenges reported by blind users and content creators with

disabilities. This problem has been investigated using an adapted version of the methodology applied to AER issue detection. NLP techniques can be used to extract and analyze frequently appearing terms—considered as potential usability-related keywords—from scientific literature discussing usability issues in Android applications. A word cloud was generated using standard NLP processing. With the help of modern AI models and their integrated word embedding representations, a clustering technique was applied to group semantically related terms, revealing categories of usability problems such as internationalization and audio-related issues. A workflow similar to the AER identification methodology can be designed, incorporating NLP techniques and AI-based similarity measures such as cosine similarity to detect additional relevant terms in versioning platforms like GitHub. Using a dataset consisting of approximately 35 applications and 65 research papers, several terms with strong semantic value for usability issue detection were identified. Clusters were then formed to categorize different types of usability concerns. Based on this, a dataset of GitHub commits can be created to investigate traces of usability problems in commit messages, allowing their detection through NLP techniques.

Chapter 10

Conclusions and Future Work

We have explored the problem of AER issues with a focus on Android applications. First, we identified the phenomenon and examined the proposed solutions from other areas of software engineering, defining research questions to analyze this problem within the Android context. Afterwards, we implemented and adapted multiple solution approaches for the Android environment. For each approach, we analyzed and compared results according to the experiments discussed in the related work. Finally, we outlined possible extensions of the proposed methodologies for identifying AER issues, including opportunities to examine other software architecture quality attributes and to detect problems present in specific architectural layers (such as the UI layer of an Android application). With this work, we can present a conclusion aligned with the main objective of the research and the initial research questions.

10.1. Research Questions conclusions

- **How can we identify Architectural Erosion in Android apps?** Architectural erosion can be identified in different ways. The most well-known identification methodologies of AER are powered by static code analysis tools and the use of AI models and NLP techniques. With these methodologies, we can identify some issues and architectural violations related to one or more architectural guidelines and standards. For AER issues in Android applications, we can identify them with code analysis of the source code of any Android project. We can identify any issue with custom lint check rules or the evaluation of some

AI model with code issues detection as a learning task.

- **What methodologies can we use to detect AER in Android apps?** The research examined two primary methodologies for AER issue identification. The first approach focused on static code analysis. A dataset of GitHub commits from several Android applications was analyzed, and issues were manually tagged by two judges experienced in Android development. From this manual process, a set of recurring patterns was identified based on architectural smells and detected issues. These patterns were then transformed into custom lint check rules integrated into the Android Studio linter. The rules were tested on multiple applications, producing warnings and error reports that demonstrated their ability to detect architectural violations. The second methodology relied on AI models and NLP techniques, using both static and dynamic word embedding models to identify potential keywords within GitHub commit messages that could indicate AER issues in the corresponding code. Multiple embedding models were compared to determine which produced keywords with the highest semantic relevance in the context of AER. Additionally, AI models were fine-tuned specifically for the task of AER issue detection, and their results were evaluated using another set of commits alongside an AI agent to measure accuracy. Overall, both methodologies delivered strong results in Android projects and demonstrated potential to assist in identifying issues related to specific software quality attributes.

- **How effective are the proposed methodologies?** The implemented methodologies for AER issues identification were tested in different ways. The methodology based on static code analysis techniques was tested with two applications: one that includes some anti-patterns related to architectural guidelines for Android projects, and another that was part of the selected set of applications used for GitHub commit extraction. In both cases, the custom lint check rules generated reports and warnings in the IDE during development, demonstrating good performance. These results suggest that additional lint rules could be created to perform deeper searches for architectural smells within the large set of explored commits. The effectiveness of the second solution methodology was evaluated by comparing it to other AI model testing approaches, manual tagging, and the use of AI agents. The testing results were promising, showing good performance in manual tagging and opportu-

nities for improvement in the AI agent-based evaluation. Overall, the effectiveness of the solution methodologies for identifying AER issues enables the expansion of this research topic and opens the door to exploring additional methodologies connected to architecture guidelines and various quality attributes.

According to the answers to the research questions, it is possible to analyze and compare the results obtained from every solution methodology. We defined a set of results and conclusions according to the testing stage of every methodology.

10.2. Conclusions of explored methodologies

10.2.1. Static Code Analysis techniques

There are some important results related to the development of custom lint check rules for AER issue identification. The built tool for the research scope showed advantages for AER issues:

- Static code analysis is very important to identify patterns of different kinds in terms of any programming language components. We can use this approach to identify any issues or insights related to any specific pattern during the development stage. The use of custom lint check rules could support the validation of different rules and standards of different frameworks.
- For Android applications, we can use custom lint check rules to detect customized patterns according to any defined pattern in the design stage during the software development cycle. Based on the tools and solutions mentioned in the related work, we can adapt them to solve customized issues in source code fragments of any Android project.
- It is possible to combine different solution approaches to improve the performance of static code analysis techniques. The use of NLP techniques and word similarity metrics could be useful to identify patterns and implement them with custom lint check rules. This approach could be improved with the current tools of AI models and NLP techniques.
- Static code analysis must be a very important step in the testing and maintenance stages of the software development cycle. The detection of general and customized lint check rules

could improve the code performance and the sustainability of any software project. For Android projects, it could improve the costs of development, testing, and maintenance. Use of static code analysis techniques in Android projects would improve the user experience, detecting different issues and patterns in different layers of the application.

10.2.2. NLP techniques and AI models

Current tools of AI and NLP can improve at a high scale for detecting anti-patterns and identifying AER issues. The use of static and dynamic word embedding models, AI models, and AI agents defined a complete tool suite to explore different insights inside the source code fragments of Android projects.

- Both static and dynamic word embedding models can be useful for identifying potential keywords in GitHub commits from Android projects. Static word embedding models rely on the training context, and when trained within a software development domain, they can yield more accurate results by uncovering new keywords related to quality attributes in Android applications. To measure similarity, the cosine similarity metric proves effective, as its averaged values can highlight semantically related words within a given context. On the other hand, dynamic word embedding models, often derived from advanced AI architectures, offer an even more powerful approach for detecting new keywords. Their training process allows them to adapt to specific contexts, making them well-suited for both classification tasks and the generation of new data.

- The use of AI models powered by NLP techniques could improve the detection of different issues with a correct training stage. Furthermore, the fine-tuning of AI models related to the detection of code smells as a learning task could detect in an early stage any architectural smell and identify AER issues.

- AI agents are useful as judges in the testing stage. AI agents can connect with LLMs at a high rate of processing and accuracy in terms of the detection of issues in any programming language. AI agents could be extended to make a connection with little or simplified models to get good processing and responses with a low rate of consumption of resources.

- NLP techniques are very important to detect words with a high semantic value related to any specific context. With processes like stemming, lemmatizing, and stop word removal, it is possible to find potential keywords that could represent any issue in any specific context. Since the context of versioning platforms of software like GitHub, to the context of reviews in any marketplace of applications, it is possible to measure the semantic value of any word to detect an issue in any project. Focused on Android projects, the use of AI and NLP could extract issues related to the user experience provided by the application. Current AI models will achieve the quality of code and sustainability of any Android project.

We observed prominent results with the implementations of every solution methodology for AER issues identification. With a combination of both methodologies, we can improve the identification process and generate optimal solutions to any identified AER issues in the source code of any Android project. Furthermore, with the mentioned methodologies extensions, we can define a prominent future work with a growth opportunity for the detection of any issues related to general or specific quality attributes. The main objective is to extend not only in the Android development environment, but extend them to the identification of issues of any software project based on different architectural patterns and guidelines, and written with any programming language.

10.3. Future Work

In the extensions section, we defined initial steps to apply the studied methodologies for AER issue identification to detect problems in any quality attribute, specifically usability issues. Using the generated commit dataset, it is possible to train machine learning models to evaluate various metrics related to detecting architectural erosion (AER) issues, particularly in Android application development. Additionally, large language models (LLMs) such as CodeReviewer and CodeT5, which are trained for code issue detection and high-performance code generation, can be adapted to tasks such as instruction-based code generation, providing feedback on input code, and producing optimized versions of the source code [**? ? ?**]. Based on the results obtained from implementing static word embedding models, dynamic word embeddings—integrated into large language models (LLMs) trained for software analysis and generation—may offer improved effec-

tiveness. Models such as CodeReviewer and CodeBERT, which leverage dynamic embeddings, can more accurately identify semantically meaningful keywords in commit messages, especially those associated with architectural erosion (AER) issues in source code. By using these models and their ability to capture contextual semantics, it becomes feasible to generate code that adheres more closely to architectural guidelines and standards. While AI systems are already capable of producing complete software projects, incorporating issue-detection mechanisms could significantly enhance the quality and robustness of the generated software. Today, there are many AI models and trained word embedding models for different training tasks. Using these models can improve the quality of the software development process. It is possible to extend the implemented methodology to learning tasks in Large Language Models (LLMs), such as creating AI agents for keyword detection related to specific quality attributes. This research could help analyze large amounts of data in Android projects to evaluate architecture quality. These solutions could reduce costs in any software project. It is essential to leverage process optimizations at every stage of software development, and the use of AI and NLP could ensure better development quality and adherence to standards and policies for a specific architecture [**? ?**].

Chapter A

TITLE

Bibliography

The references are sorted alphabetically by first author.

[1] Guide to app architecture. *Google*, 2023.

[2] Detekt. *Detekt team*, 2025.

[3] F-droid. *F-Droid*, 2025.

[4] Sonarlint. *Sonar*, 2025.

[5] D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N Natarajan. Masai: Modular architecture for software-engineering ai agents. *Arxiv*, 2024.

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

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

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

[9] CodeAcademy. Abstract syntax tree. *CodeAcademy*, 2024.

[10] O. Dabic, E. Aghajani, and G Bavota. Sampling projects in github for msr studies. *18th IEEE/ACM International Conference on Mining Software Repositories*, 2021.

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

[12] GENSIM. Gensim topic modeling for humans. *GENSIM*, 2024.

[13] N Gulshan. Understanding mvc, mvvm, and mvp: A comprehensive comparison. *Medium*, 2023.

[14] M. Jean de Dieu, P. Liang, M. Shahin, C. Yang, and Z Li. Mining architectural information: A systematic mapping study. *Springer link repository*, 2024.

[15] S. Juneja, A. Nauman, M. Uppal, D. Gupta, R. Alroobaea, B. Muminov, and Y Tao. Machine learning-based defect prediction model using multilayer perceptron algorithm for escalating the reliability of the software. *ACM Digital Library*, 2023.

[16] 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.

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

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

[19] P. Liang, P. Avgeriou, and L Ruiyin. Towards automated identification of violation symptoms of architecture erosion. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2024.

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

[21] J. Pennington, R. Socher, and C Manning. Glove: Global vectors for word representation. *University of Stanford*, 2014.

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

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

[24] Li Ruiyin. Understanding, analysis, and handling of software architecture erosion. *University of Groningen*, 2023.

[25] T. Sivayoganathan and M Ramzan. Evaluating webpage performance and web accessibility of canadian universities' mental health service webpagess. *Spring Nature*, 2024.

[26] D Spadine. Pydriller. *PyDriller*, 2018.

[27] Sphinx. Nltk. *NLTK*, 2023.

[28] L. St. Amour and E Tilevich. Toward declarative auditing of java software for graceful exception handling. *ACM Digital Library*, 2024.

[29] S. Staroletov. A method to verify parallel and distributed software in c sharp by doing roslyn ast transformation to a promela model. *ResearchGate*, 2019.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I Polosukhin. Attention is all you need. *Arxiv*, 2017.

[31] D. Wang, Y. Zhao, S. Feng, Z. Zhang, W. Halfond, C. Chen, X. Sun, J. Shi, and T Yu. Feedback-driven automated whole bug report reproduction for android apps. *ACM Digital Library*, 2024.

[32] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J Lou. Large language models meet nl2code: A survey. *Arxiv*, 2023.

Acronyms