MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

AI-based Anti-patterns Detection in Java Software Enterprise Development

Master's Thesis

MARTIN PROCHÁZKA

Brno, Spring 2026

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

AI-based Anti-patterns Detection in Java Software Enterprise Development

Master's Thesis

MARTIN PROCHÁZKA

Advisor: doc. Bruno Rossi, PhD

Department of Computer Systems and Communications

Brno, Spring 2026


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source. During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar check and improving my writing style,
- GitHub Copilot for faster code, test writing and documentation.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Martin Procházka

Advisor: doc. Bruno Rossi, PhD

Acknowledgements

I would like to express my sincere gratitude to my advisor, Bruno Rossi, for his time, guidance, and valuable insights, which greatly contributed to the completion of this thesis.

I would also like to thank my family for their continuous support, patience, and encouragement throughout my studies.

Abstract

Software quality is often negatively affected by the presence of anti-patterns in codebases. They represent suboptimal solutions that often lead to maintainability, scalability, and performance problems. These issues are more important in enterprise software development.

This work aims to extend JENA (Java Enterprise Analyzer), introduced in previous theses. JENA is a static analysis tool that detects anti-patterns in Java Enterprise software using rule-based detectors. This work adds the ability to detect enterprise anti-patterns using machine learning solutions. We also introduce a new, refactored version of the tool, better deployment and infrastructure for the machine learning evaluation. The new version of JENA is called Java Enterprise Anti-patterns Detector (JEAD).

The machine learning detectors implemented in JEAD were evaluated on open-source projects. The results show that they have great potential for the detection. In some cases, they generalise better across codebases without requiring manual threshold adjustments typical of rule-based detectors. But compared to them, they require greater effort for training and deployment, increasing the time needed for analysis.

Keywords

Static analysis, Java, Spring Boot, Machine Learning, Artificial Intelligence, Software Quality, Software Architecture

Contents

1 Introduction **1**
- 1.1 Objectives of the Thesis . . . 2
- 1.2 Thesis Structure . . . 2

2 Software Quality, Patterns and Anti-Patterns **3**
- 2.1 Software Quality . . . 3
- 2.2 Software Quality Factors . . . 4
- 2.3 Software Metrics . . . 5
- 2.4 Patterns in Software Development . . . 6
- 2.5 Software Anti-patterns . . . 7
- 2.6 Code Smells and Anti-patterns . . . 9
- 2.7 Usage of AI/ML in Software Quality . . . 10
  - 2.7.1 Automatic Code Refactoring . . . 10
  - 2.7.2 Explainable Software Defect Prediction . . . 12

3 Related Works **14**
- 3.1 Enterprise Anti-patterns Catalogue . . . 14
- 3.2 State of the Art in Anti-pattern Detection . . . 15
  - 3.2.1 Rule-based Approaches . . . 16
  - 3.2.2 Machine Learning Approaches . . . 18

4 JEAD: Java Enterprise Anti-patterns Detector **23**
- 4.1 Previous Version – JENA.V2 . . . 23
  - 4.1.1 Space for improvements . . . 24
- 4.2 Machine Learning Detectors . . . 25
  - 4.2.1 Architecture . . . 26
  - 4.2.2 Code Extractors . . . 26
  - 4.2.3 Inference Queues . . . 28
  - 4.2.4 Communication with Inference Server . . . 29
  - 4.2.5 YAML Configuration . . . 30
- 4.3 Additional Work . . . 31
  - 4.3.1 Lombok Support . . . 32
  - 4.3.2 Dependency Updates & Migration to Java 25 . . . 34
  - 4.3.3 Improved Project Structure . . . 35
  - 4.3.4 Enhanced Deployment . . . 36

5 Inference Server **38**
5.1 Existing Solutions 38
5.2 Architecture 39
5.3 Project Structure 41
5.4 Public gRPC Interface 41
5.5 Models Root Folder Structure 42
5.6 Models Loading and Communication 44
5.7 Dockerization and Deployment 45
5.8 Future Work 46

6 Detecting anti-patterns Using Machine Learning Models **47**
6.1 Anti-patterns Selection Motivation 47
6.2 (DI8) God Dependency Injection Detection 48
6.3 (SER1) Multi-service Detection 50
6.4 (TECH_DEBT) Technical Debt Detection 52

7 JEAD Evaluation on Open Source Projects **56**
7.1 Project Selection Criteria 56
7.2 Evaluation Strategy 57
7.3 (DI8) God Dependency Injection Anti-pattern 58
7.4 (SER1) Multi-service Anti-pattern 59
7.5 (TECH_DEBT) Technical Debt Anti-patterns 61
7.6 Validity and Applicability of the Models 63
7.7 Threats to Validity 66

8 Conclusion **67**

Bibliography **69**

A Java Enterprise Anti-pattern Catalogue **78**
A.1 Dependency Injection 78
A.2 Persistence Layer 80
A.3 Service Layer 81
A.4 Mocking 82
A.5 Security 83
A.6 Technical Debt 84

B JEAD Project Structure **86**

C Running the Evaluation **87**

D Architecture Diagrams **89**

E Electronic Attachment **92**

List of Tables

2.1 Examples of code metrics used for code smell detection [9] 6
2.2 Examples of Artificial Intelligence (AI) usage in Software quality 13
3.1 Overview of anti-patterns used in this thesis, based on Mička [4] and extended with Technical Debt 15
3.2 Related rule-based anti-pattern detection tools 19
3.3 Related machine learning anti-pattern detection approaches 22
4.1 JEAD architecture concerns description 27
4.2 List of selected updated dependencies 35
5.1 Existing Inference Server solutions 39
6.1 Overview of Machine Learning (ML) anti-patterns detectors with technologies used 48
6.2 Projects used for technical debt model fine-tuning 54
7.1 Confusion matrix of DI8 anti-pattern detection per project 59
7.2 DI8 anti-pattern detectors performance measures comparison (project Klaw) 59
7.3 DI8 anti-pattern detectors performance measures comparison (project Apollo) 60
7.4 Confusion matrix of SER1 anti-pattern detection per project 61
7.5 SER1 anti-pattern detectors performance measures comparison (project Klaw) 61
7.6 SER1 anti-pattern detectors performance measures comparison (project Apollo) 62
7.7 True and False positives statistics of TECH_DEBT model . 63
7.8 TECH_DEBT model performance on test dataset during training 64

List of Figures

4.1 JEAD Machine Learning Detection Flow 25
4.2 Structure of the JEAD zip package 37
5.1 Structure of the Inference Server project 41
5.2 Example of models root folder structure 43
5.3 Simplified architecture of Inference Server with models root folder 44
6.1 Multi-service detector machine learning pipeline 52
D.1 System-level diagram of JEAD architecture 89
D.2 Component-level diagram of JEAD architecture 90
D.3 Component-level diagram of Inference Server modular architecture 91

1 Introduction

Every codebase ever written contains many patterns. Patterns clarify how components interact, making it easier for developers to collaborate and understand the system better. They also facilitate easier system refactoring while aiming to minimise the necessity for such changes. [1]. These patterns are considered good practice. However, many developers work on a codebase over its lifetime, so it might be touched by developers with different areas of expertise and seniority. Lack of codebase overview, knowledge, or experience may result in the creation of patterns that might be considered poor design decisions or bad practices. Such decisions might lead to reduced maintainability, scalability, and extendibility of the system and also have a significant impact on software quality. They can be easier to implement and use in the short term, but harder to maintain in the long term – we can consider such patterns anti-patterns [2].

Such patterns can be used in high-level architectural decisions as well as in smaller-scale decisions, for example, at the class level. Enterprise development primarily includes large-scale applications used by large companies, so the anti-patterns may appear at all levels of system design. To mitigate the risk of introducing anti-patterns in enterprise applications, it is necessary to use an analysis tool that helps software engineers discover them. This tool, called JENA (Java Enterprise Analyzer), was introduced in the thesis *Catalogue of Anti-Patterns in Java Enterprise Development* [3] and then improved in the following work *Anti-patterns Detection in Java Software Enterprise Development* [4]. Those theses also introduced a catalogue of Java enterprise anti-patterns describing 28 bad practices divided into 5 groups.

JENA is a static analysis tool that takes source code as input, parses it into an Abstract Syntax Tree, and extracts code metrics or patterns. They are then evaluated against statically defined thresholds in the configuration file. If a value exceeds defined thresholds or does not meet the condition, it is presented to the end user as an issue (detected anti-pattern). Such a rule-based approach might lead to many false positives and cannot dynamically adapt to the specific codebase.

1.1 Objectives of the Thesis

This thesis aims to analyse the feasibility of using machine learning to detect Java Enterprise anti-patterns. The machine learning models could potentially help mitigate the problems mentioned in the previous paragraph. They could improve detection by learning dynamic thresholds for code-metric-based anti-patterns, understanding the semantics of the text to detect anti-patterns based on method names, or finding semantic patterns in the code, for example, `TODO`-like comments.

The primary goal is to identify multiple anti-patterns and develop a prototype of ML models to detect them. The thesis introduces a refactored version of JENA.V2 that adds compatibility with ML detectors. With that, we present a simple and lightweight solution for deploying models. In the end, the thesis presents an evaluation of ML models on open-source projects, comparing them with rule-based detectors.

1.2 Thesis Structure

Firstly, Chapter 2 discusses topics in software quality, anti-patterns, and the use of AI in software quality. Then, Chapter 3 briefly summarises the Enterprise Anti-pattern Catalogue, the current state of the art, and solutions for utilising machine learning models for detection. Chapter 4 introduces a new version of JENA, called JEAD, that allows sending code snippets to machine learning models for evaluation. It also introduces new features, such as Project Lombok support [5], and improved deployment to enhance the user experience. Chapter 5 presents a lightweight service for deploying machine learning models for inference with a modular and plugin-based architecture. Chapter 6 shows three prototype models trained to detect three anti-patterns. Each anti-pattern is detected using a different type of model and approach. Last but not least, Chapter 7 presents the evaluation results for the ML models on two open-source projects, with a clear overview and comparison with rule-based detectors.

2 Software Quality, Patterns and Anti-Patterns

This chapter summarises basic terms in software quality. First, it discusses how software quality is defined, the quality models, software metrics, design patterns, and how to recognise anti-patterns. The last section discusses the use of AI in software quality with multiple examples of state-of-the-art work in this area.

2.1 Software Quality

There is no single definition of the term *software quality*. Gaffney [6], one of the first researchers to discuss software quality, defines it as a reproducible mathematical measure composed of various software characteristics. The "goodness" or "badness" of the measure is defined by pre-established thresholds. Software metrics can tell how reliable and maintainable the software is. They can also reveal flaws in the development process – for example, when the metric like *test coverage* is low, it can mean that the process places very little emphasis on writing new tests.

The quality of the software does not consist only of measurable metrics. A recent study among Google employees [7] presents the theory that quality is not just about metrics; it comprises four distinct, interconnected types of quality. Those types include:

Process quality

It refers to the quality of the development processes. Google's theory is that a high-quality software product starts with a high-quality development process. This can include comprehensive testing, code reviews, and effective planning.

Code quality

The Google developers described the code quality as "the ease of working with and understanding the code". The easier it is to work with code, the lower the developer's mental overhead when reading it. Thus, it can lead to increased maintainability, readability, developer

velocity, and overall product quality. This is the primary quality aspect on which this thesis will focus.

System quality

The system quality refers to the "quality as the business sees it". It often refers to metrics such as performance, security, outages, and the product's behaviour in the production environment. It is a challenge to measure this type of quality, since events like outages or security incidents are rare – there might not be enough data to statistically prove improvements or deterioration.

Product quality

This type of quality is primarily defined as "how the customer experiences it". It refers to the product's usability, reliability, and end-user experience.

2.2 Software Quality Factors

McCall et al. defined one of the first software quality factors [8]. They define a factor as "a condition or characteristic that actively contributes to the quality of the software". The factors might include developers, users, and business needs. They divided factors into multiple groups, some examples are:

- correctness – satisfaction of requirements,
- maintainability – effort to fix errors in the code,
- reusability – the program can be used in other applications,
- testability – effort required to test if the program does its intended function,
- usability – the user's effort to enter inputs and interpret outputs of a program.

Each factor has assigned criteria, which are attributes by which the factor can be identified – for example, factor *efficiency* includes criterion *storage efficiency* defining the minimum storage requirements during a particular operation. Each criterion includes quantitative metrics. In the case of *storage efficiency*, one metric is the storage requirement for the system to be allocated. This hierarchical structure defines a comprehensive framework for software quality metrics in relation to quality factors.

2.3 Software Metrics

Software metrics are often used to measure the quality of software systems. This allows managers, stakeholders and developers to track quality and a quantitative overview of their system's complexity. They are also often used to detect particular code smells or anti-patterns. There are multiple studies and standards defining them. Palomba et al. [9] introduce a bug prediction framework based on the code smells detected by the code metrics. Table 2.1 shows an example of code metrics used for the detection of particular code smells. Based on the metrics and code smells detected, they assess their severity to predict software bugs. Chidamber and Kemerer created one of the most famous metric sets, known as CK metrics [10]. They focus on measures in object-oriented systems and include 6 metrics that measure class, inheritance, and coupling complexity.

There are also more straightforward metrics, such as the number of methods and fields in a class, the number of classes, and the number of levels in the class hierarchy [11]. Each metric can capture different details of the code structure and complexity, and might be used to measure different aspects of the code.

Even standards from the ISO family define some sets of metrics. For example, the ISO/IEC 9126 [12] family of standards divides software metrics into internal and external. External metrics measure software behaviour during system use and can be used during the testing and operations phases of the product lifecycle. Otherwise, internal metrics are typically used during the development stage. The metrics are divided into categories like Security metrics, Usability metrics, Maintainability metrics, etc. Each category has its representative in both

Table 2.1: Examples of code metrics used for code smell detection [9]

Metric	Description	Used for
Access To Foreign Data (WMC)	The number of attributes from unrelated classes accessed by accessor method.	God Class
Changing Classes (CC)	Number of classes which calls measured method	Shotgun Surgery
Changing Methods (CM)	Number of methods which call measured method	Shotgun Surgery
Lines of Code (LOC)	Number of lines of codes of method / class	Brain Method
McCabe Cyclomatic Complexity (CYCLO)	Number of linearly independent paths in a method.	Brain Method
Number of Accessor Methods (NOAM)	The number of getter and setter methods.	Data Class
Weight of Class (WOC)	The number of public methods divided by the total number of public members.	Data Class

external and internal metrics. The ISO/IEC standard is today replaced by the ISO/IEC 25002:2024 family ISO/IEC 25002:2024 [13], which defines software quality requirements and their evaluation.

2.4 Patterns in Software Development

Patterns are the basic elements of software design. In general, they describe how components are structured, how they communicate, how they distribute responsibilities, and what their roles are [1]. They serve as the building blocks of software components. Developers often need to solve problems they have experienced in the past, but at the core, the solutions are very similar. Software patterns abstract those solutions and make them reusable for similar cases. Patterns also define implementation skeletons that help developers to easily implement the desired functionality. Nevertheless, they help to build larger software systems with complex architectures. Software patterns are often grouped into 3 categories: design, architectural patterns, and idioms [14].

Design patterns are often used with object oriented design; we are talking about relationships between classes or objects [1]. The application of design patterns is "lower level" than that of architectural patterns, and their application does not often influence the architecture of the entire system, only a minor part (subsystem) [14].

Architectural patterns commonly serve as foundational structural templates for software architectures. They have a significant impact on how subsystems are structured and how they interact. These patterns might define the internal structure (architecture) of an individual subsystem (folder structure, responsibilities, ...), or the composition and structure of multiple related subsystems. The selection of appropriate architectural patterns is crucial for achieving a maintainable and scalable design for software systems [14].

Idioms, unlike the previous two categories, are specific to a particular programming language. The context of one idiom might be shared across languages, but the actual implementation details are different. They represent the lowest level patterns [14]. Among other things, their purpose is to define best practices for writing clean, dynamic code specific to the programming language. An example of such an idiom could be the naming convention – in Java, method names are often written in `lowerCamelCase()`, but in Python, words are separated with underscores – `method_name()`.

2.5 Software Anti-patterns

The anti-patterns are in the core very similar to the software patterns described in the previous section. The main difference is that patterns are often considered best practices – they often bring many benefits, such as improved maintainability, scalability, and code quality. In contrast, anti-patterns are bad practices. According to Brown et al. [2], it is "a common solution to a problem that generates decidedly negative consequences". With frequent use of anti-patterns, software quality may decrease, code may become unreadable or untestable, and any change to the codebase may require significant effort. An anti-pattern can be created from a properly used design pattern, but in an unsuitable place in the code, due to the programmer's lack of knowledge

or experience [2]. The anti-patterns can also be divided into design and architectural levels, similarly to the patterns mentioned in Section 2.4.

The introduction of anti-patterns into the code base is almost never accidental. It often arises from uninformed decisions, time pressure, or a lack of knowledge or experience. William et al. identified multiple mistakes that cause the creation of anti-patterns [2]:

- **Haste** – making hasty decisions often leads to trade-offs and a decrease in software quality. The typical victim is testing.

- **Apathy** – often causes us unwillingness to solve known problems and bugs. Not all problems have straightforward solutions, but a passive approach to solving them might lead to the adoption of improper solutions.

- **Narrow-mindedness** – refusing to learn new best practices and standards. Developers often continue to work the same way they have for years, showing little willingness to adapt or change their approach.

- **Sloth** – a sign of a developer who consistently chooses the first and easiest solution without carefully considering its implications or long-term consequences.

- **Avarice** – architecture complexity often leads to many challenges in maintainability and development of a software system. Under time pressure and fast time-to-market, some development steps might be skipped (such as testing). This can lead to significant problems for the system's future.

- **Ignorance** – defined as "intellectual sloth" – the failure to seek understanding. This results in people who are unaware of how to use technology properly.

- **Pride** – developers often prefer to invent new designs of solutions, instead of reusing existing ones. Such an approach might lead to an increased need for testing and documentation, adding unnecessary complexity.

Example of Design vs. Architectural Anti-patterns

God Class	Big Ball of Mud
Design Anti-pattern Class with too many responsibilities. It heavily accesses data from other layers, is large and complex, and has very low method cohesion. In a well-designed system, responsibilities should be uniformly distributed among classes [15].	*Architectural Anti-pattern* It occurs when there are no defined boundaries between subsystems, with no regulated growth. The overall system's structure is not well defined. One change could break multiple dependent functionalities due to tight coupling [16].

2.6 Code Smells and Anti-patterns

Many sources use this term interchangeably. They are tightly coupled, degrade software quality, increase technical debt, and reduce the maintainability of software systems or codebases. But each of them affects software quality at different levels. According to Rivera et al. [17], the anti-patterns are defined more on the design level of a software system. They are often categorised as *Linguistic Anti-patterns*, *Software Reuse Anti-patterns*, *Performance Anti-patterns*, *Scrum Anti-patterns*, ... Code smells are often defined at the lower, implementation level. They often refer to code structure issues, long methods, database smells, or presentation smells. Code smells can often be eliminated through refactoring, while anti-patterns represent deeper structural problems and may require greater effort to understand and eliminate. But there is no strict difference between those terms, and some anti-patterns (or code smells) may partially belong to both categories.

We can demonstrate this interchangeability and no strict difference on the Enterprise Anti-patterns mentioned in catalogue by Mička [4]. For example, anti-pattern DI10 (Long Producer Method) can be seen as a code smell because it indicates method complexity at the implementation level. Also, the anti-pattern DI8 (God DI Class) mentions the number of injected dependencies – this might belong to both categories, since it can be understood simply as the number of injected classes through the constructor (in other words), which is

an implementation-level issue. Or it can be seen as a design-level issue, where responsibility is incorrectly spread across classes, and some classes implement more functionality than they should.

2.7 Usage of AI/ML in Software Quality

This section describes various approaches and tools that leverage multiple forms of AI to improve the quality of software systems. Traditional rule-based systems make decisions based on predefined thresholds. Therefore, they lack automatic adaptability to new environments and the ability to improve over time. AI models can learn dynamic thresholds, adapt to changes, and perform more complex tasks.

The following text highlights only a few topics, while the actual scope of AI application in software quality is considerably broader.

In the context of this thesis, we use traditional supervised learning algorithms, such as Random Forests. For semantic understanding and classification, we use Natural Language Processing (NLP) processing models like CodeBERT. These models belong to the Deep Learning category of Machine Learning. For clustering, we use unsupervised machine learning algorithms like HDBSCAN. Since all of these algorithms and approaches fall under the category of Machine Learning in Artificial Intelligence, we refer to the term simply as ML for clarity.

2.7.1 Automatic Code Refactoring

Refactoring is the process of improving an existing codebase. It is a series of small modifications to existing code whose primary purpose is to improve the quality of the refactored system by enhancing its structure, testability, performance, or readability. Refactoring should preserve the code's functionality. It is essential to have refactored code covered by tests, so that developers minimise the risk of breaking functionality and introducing new bugs.

The most common is manual refactoring. The developer performs it manually using code suggestions and tools from the Integrated Development Environment (IDE), the compiler, and unit tests, which can be run after each refactoring step. This approach might require a high effort, especially in large enterprise-grade systems. Also, use the

static analysis and refactoring assistance tools integrated into the IDE (such as IntelliJ IDEA). Those tools are quite powerful but do not offer a deeper semantic understanding of the code or the ability to identify its performance issues [18].

Refactoring for performance optimizations

To improve automatic refactoring capabilities, Polu [18] suggested an AI-driven refactoring framework for performance optimisations. The tool utilises multiple methods and approaches. First, it parses the code into an Abstract Syntax Tree (AST), so the AI model can extract structural and semantic information (e.g., function calls, ...), and then converts it to a Graph Neural Network (GNN) representation. Using transformer-based models, such as CodeBERT, it analyses the code semantics and provides refactoring suggestions. In the end, it runs the performance evaluation on both the original and refactored code to see the results. The resulting framework successfully optimises the refactored code with 98% accuracy.

Refactoring using Large Language Models (LLMs)

The use of LLMs shows great potential in code refactoring tasks. However, they still introduce many hallucinations and need human intervention or verification of the suggested output. Cordeiro et al. [19] conducted research on current opportunities and limitations. They identified current approaches that use LLMs, such as Test-Driven Development (TDD) and automated code refactoring. Regarding refactoring, models integrated into IDEs often suggest changes while the developer is doing the manual work based on the suggestions, because they lack contextual understanding. The mentioned studies also show that the models perform much better when the prompt includes an explanation of refactoring categories, which the model should look for. In the end, the study suggests that a closer integration of automated test generation is needed to improve LLM refactoring capabilities. Developers can generate tests based on the modified code when no existing test case is present. Another research focus should be on context-enriched prompts to provide more context for refactoring

code into LLMs. They suggest including more details in the prompts, such as a guide to remove specific code smells.

2.7.2 Explainable Software Defect Prediction

Before releasing a software system to production, stakeholders want to be sure it is reliable, well-tested, and has few weaknesses. A method of software defect prediction (SDP) enables analysis of the codebase, pinpointing weak points and helping allocate appropriate testing resources.

Geçer et al. [20] developed a framework that uses ML models to predict software defects. Since ML models are often "black boxes" with limited transparency, stakeholders may not always trust their outputs. Therefore, they used a method called Explainable AI (XAI). It enables model interpretability and reasoning about the model's internal structures. Global explanations enable visualisation of the importance of features and the overall model's logic, while local explanations provide insights into the concrete decisions made (for example, why the following module was marked as defective). The whole framework utilises multiple ML models (random forests, naive Bayes, neural networks, and multilayer perception), trained on a large benchmark data set containing 21 software metrics. The solution helps developers, managers, and stakeholders prioritise the efforts needed for defect mitigation, together with explanations of concrete metrics that influence quality.

Table 2.2: Examples of AI usage in Software quality

Topic	Technology
AI-driven refactoring framework for performance optimisations leveraging neural networks and transformer-based models [18].	AST, GNN, CodeBERT
Research suggestions on using LLMs in refactoring and development, mainly using patterns like TDD [19].	LLM, TDD, IDE integration
Using Explainable AI models to increase trust in the models by stakeholders during defect prediction [20].	XAI, Random forests, Naive Bayes, metrics
Overview of usage AI in Quality Assurance for test data synthesis, test case generation, exploratory testing, defect tagging [21].	LLMs integrated in IDE, DNN, NLP

3 Related Works

The first part of this chapter discusses the catalogue of enterprise Java anti-patterns introduced in the previous thesis [3][4], providing a clear overview of the included anti-patterns. The second section focuses on state-of-the-art work on anti-pattern detection in software development.

3.1 Enterprise Anti-patterns Catalogue

While many internet sources refer to catalogues of anti-patterns, only a few are related to enterprise development. Previous thesis *Catalogue of Anti-Patterns in Java Enterprise Development* by Bateľ [3] introduced a set of 24 anti-patterns related to enterprise applications in Java. The catalogue is divided into 5 categories: General Issues, Persistence Layer, Service & facade Layer, Dependency Injection, and REST API.

The follow-up thesis *Anti-patterns Detection in Java Software Enterprise Development* by Mička [4] revised the first version of the catalogue. More generic anti-patterns (like unused imports or variables) were removed, while adding a set of new, strictly enterprise-related anti-patterns. The new version of the catalogue is also divided into 5 categories with a total of 28 anti-patterns. It introduces new categories, such as Security and Mocking, while removing non-enterprise-related categories, such as General Issues or Representational State Transfer (REST) Application Programming Interface (API), from the first version.

The catalogue's primary purpose is to help developers learn about them, since those anti-patterns might cause serious performance, maintainability, and security problems. It also serves as a template for an automatic detection tool called JEAD (formerly called JENA, introduced in [3]), which is capable of detecting these issues using a rule-based approach, and also using machine learning (ML) models (see Chapter 4).

The brief summary of the catalogue from Mička [4] is in the Appendix A. This thesis introduces a new category, Technical Debt, which is presented in Appendix A.6. It also includes a brief analysis of the de-

Table 3.1: Overview of anti-patterns used in this thesis, based on Mička [4] and extended with Technical Debt

Dependency injection DI1 Unused injection DI2 Direct container call DI3 Concrete class injection DI4 Open window injection DI5 Framework coupling DI6 Multiple forms of injection DI7 Open door injection DI8 God DI class ★ DI9 Multiple assigned injections DI10 Long producer method ★ DI11 Intransigent Injection ★ DI12 Static dependence provider	**Service layer** SER1 Multi-service ★ SER2 Tiny service ★ SER3 The client completes a service ★
	Mocking MOC1 Final method call with exception MOC2 Constructor call with exception MOC3 Static Call With Exception MOC4 Inappropriate Method Mocking
Persistence layer PER1 N+1 query problem PER2 Dredge ★ PER3 Crush	**Security** SEC1 Storing Secrets In Insecure Places ★ SEC2 Disabling CSRF Protection SEC3 Lifelong Access Tokens SEC4 Insecure Default Configuration SEC5 Signing Jwt With Fixed Secret SEC6 Insecure Communication
Technical debt TECH_DEBT1 Self Admitted Technical Debt ★ TECH_DEBT2 Commented Out Code ★	

Symbol ★ indicates anti-patterns detectable by machine learning.

tectability of each anti-pattern by ML algorithms in the context of this thesis. Table 3.1 shows a clear overview of the included anti-patterns.

3.2 State of the Art in Anti-pattern Detection

This section describes currently available tools and research on anti-pattern or code-smell detection in source code. The literature reviewed primarily covers non-enterprise development research tools, with very few papers focusing on enterprise development. The majority of tools focus on detecting more generic anti-patterns (god classes,

large classes, ...). The chapter also discusses the differences between rule-based and machine-learning approaches to detection.

3.2.1 Rule-based Approaches

The traditional way of detecting an anti-pattern is a rule-based approach. Often based on code metrics, together with defined static thresholds. If the threshold is exceeded, the evaluated component (class, method, etc.) is marked as "bad", otherwise as "good". It is often a challenge to define static thresholds, and you need an experienced analyst to analyse them for the specific code metric. A study by Kecia et al. [22] analysed more than 40 open-source Java projects to define reference thresholds for six code metrics. They used statistical methods to find the distributions of each metric between all projects. From that, they defined reference ranges and assigned metric values as "good", "regular", or "bad". For example, for the metric "number of public methods", the good range is 0-10, the regular range is 11-40, and the bad range is greater than 40. This article shows that defining references for a single metric requires significant effort. Incorrect thresholds may lead to many false positives in detection systems. The following examples present some selected static analysis tools:

(**1**) One of the tools for rule-based analysis is the open source community-driven **PMD project** [23]. While its primary focus is detecting Java code smells and anti-patterns, it can also detect issues in 17 other languages. It is built on predefined rules; the base package includes more than 400 built-in rules. For Java, it includes rules for checking code best practices, code style, design, performance, security, etc. Customisation of the rules is done through XML configuration, where the user can easily modify pre-defined properties for each rule. It also allows the creation of custom rules, which can be implemented using Java code that directly accesses the AST or as XML configuration, where AST nodes are accessed and evaluated via XPath locators. For Java analysis, it uses the ASM framework [24] to construct the AST by reading bytecode, including runtime bytecode to resolve runtime API references. The tool builds a comprehensive abstraction over the ASM framework, which allows complete AST traversal using the visitor design pattern. It also includes a set of predefined algorithms to compute code metrics (such as the number of public methods in

a class), but the set is very limited. Thanks to this abstraction, it would be possible to create custom rules for detecting anti-patterns in enterprise development, similar to JENA.V2 (see Example 2). However, the custom rules are essentially add-ons to the core system, bundled as JAR files and included in the PMD classpath, which makes debugging the generated rules much harder.

(**2**) **JENA.V2** is a static analysis tool for detecting enterprise anti-patterns in Java. It was developed together with a catalogue (see Section 3.1) in the Master's thesis of Bateľ [3] and Mička [4]. It includes issue detectors 19 for anti-patterns listed in the catalogue. The application uses an interactive Command Line Interface (CLI) built with Spring Shell [25] as the Spring Boot application [26]. Since it is a rule-based tool, it provides a JavaScript Object Notation (JSON) configuration to set metric thresholds, regular expressions, and conditions that define rules. The tool also allows configuring which annotations are used to detect bean classes, for example. This makes it highly customizable and framework-independent. The JENA does not support adding new detectors via plugins, so whenever a new detector is added, the code must be updated. However, it provides a set of interfaces that enable easy extensibility. The tool is primarily intended for manual use via the CLI, so integration into the Continuous Integration / Continuous Delivery (CI/CD) pipeline is not straightforward. Additional functionality includes aggregating found issues and tracking the evolution of anti-patterns between project versions. It is built using the Java Parser with the Java Symbol Solver [27], enabling proper inference of Java types on the classpath. For proper work, the evaluated application must have exposed dependencies (JAR files). For this purpose, the authors suggest using Maven plugins or Gradle scripts. They also prepared a CLI command that modifies Maven Project Object Model (POM) files to include the required plugin. This thesis aims to enhance the tool through machine learning detection, while improving user experience and deployment.

(**3**) **SonarQube** [28] is the most famous, comprehensive and robust rule-based static analysis tool widely used by the largest software companies. It offers multiple tiers; the basic free community edition allows analysis of 21 languages and frameworks. The language-specific scanner is provided for each language included in the CI/CD and build process (for Java as a Maven or Gradle plugin). The scanner

then communicates with the SonarServer, which provides configurations, plugins, and other parameters needed for the scan. After the scan is completed, the scanner sends the results to the SonarServer, where users can browse an intuitive dashboard that provides statistics, scan history, and issue management. It also provides a configuration of quality gates. They include a set of conditions that the code measures during analysis. Depending on the result, the code passes or fails the quality gate. The quality gate settings are based on overall code statistics, metrics, and code issues. Like PMD, SonarQube also includes a predefined set of checks. The Java sonar scanner [29] includes more than 600 rules, of which more than 350 are code smells. The built-in checks include a lot of enterprise-level detectors, especially for the Spring framework [1]. Compared to the enterprise anti-pattern catalogue (see Section 3.1), it is missing, especially, the persistence layer and technical debt checks, but it includes many similar dependency injection rules. SonarQube supports creating plugins in Java. For the AST traversal, it uses Eclipse JDT. The plugins are then deployed to the SonarServer, which provides them to the scanners. Compared to PMD (Example 1) and JENA.V2 (Example 2), SonarQube requires a more robust infrastructure, particularly the SonarQube Server component. Alternatively, the cloud-hosted version (SonarCloud) can be used to avoid managing this infrastructure. Additionally, the Community Edition has limited functionality, and a subscription is required to access advanced features and the cloud-hosted version.

3.2.2 Machine Learning Approaches

Although rule-based approaches can sometimes achieve relatively high detection precision, finding the correct metric thresholds can be very challenging. They are also not a good choice for detecting complex anti-patterns in complex systems. Sometimes it is also necessary to capture the semantics and non-linear relationships between source code features and code smell categories, which makes it a suitable domain for machine learning (ML) algorithms. The most common is binary classification, where models learn code snippets and software metrics associated with labels ("is smell", "is clean") using supervised

1. examples: the optional REST parameter should be an object, field injection should not be used, disabled CSRF protection warning, and many more

Table 3.2: Related rule-based anti-pattern detection tools

Name	Purpose
PMD [23]	Modular rule-based detector built on top of ASM framework, primarily Java.
JENA.V2 [4]	Rule-based Java Enterprise Anti-pattern detection tool built on top of Java. Symbol Solver.
SonarQube [28]	Cloud-based enterprise solution for rule-based analysis of source code across various languages.
DIAnalyzer [30]	Java dependency injection anti-pattern analyser.
JDeodorant [31]	Eclipse plug-in for code smell detection and refactorings.

learning techniques. However, ML algorithms also include a lot of complexity and effort. The often-problem is the complex, hidden structure of the models, which might not be trusted by stakeholders and developers (see Section 2.7.2). Another pitfall is the high effort needed to collect the accurate datasets needed for learning. So, it is questionable whether the effort to train a complex model is worthwhile. The following paragraphs present a collection of machine learning tools and approaches for anti-pattern and code-smell detection.

(**1**) Ali et al. [32] introduced a novel approach to detect code smells using the relation-aware BERT (RABERT) model. The standard architectures and approaches for natural language and code processing use transformer models, such as CodeBert [33] and GraphCodeBert [34]; however, they do not perform well at capturing structural relations among various abstract components of code. The suggested approach integrates relational embeddings to capture structural dependencies between software metrics. It is especially useful for detecting structural software issues. They created a dataset based on more than 50 open-source projects from GitHub, with 20 labelled software complexity metrics, but their research primarily focuses on the Large class code smell[2]. Ali et al. conducted a comparative evaluation between transformer-based models and traditional ML techniques (gradient boosting, decision trees, ...). Among transformer models, RABERT significantly outperformed other BERT-based models with 91% pre-

2. A class accumulates many responsibilities, making it harder to maintain.

cision but lower recall (53%), resulting in an F1 score of 53%. This means their selected approach is better suited to high-assurance systems, where false positives must be minimised. Among traditional ML techniques, the model with the highest F1 score was Gradient boosting, but it had lower precision at 80% – it generated more false positives. They also compared training and inference time, showing that transformer-based models are much slower in both measures. So, classical ML models are better suited for real-time applications.

(**2**) The study by Reis et al. [35] presents an experiment using an approach called Crowdsmelling to collect datasets for training ML models to detect code smells. The experiment took 3 years, during which more than 100 teams with 3 members each classified the presence of 3 code smells (Long Method [3], God Class [4], Feature Envy [5]). This resulted in a data set that was used to train 6 ML models. The participants were third-year students in the software engineering course of the bachelor's study program. The primary objective of the research was to find out if the Crowdsmelling approach is suitable for code smell detection and what is the machine learning model. The results show that the models perform best at finding the God Class anti-pattern (F1 score around 80.5% on average across all models), then Long Method (around 76%) and Feature Envy (around 45%). The authors note that classifying more complex code smells with fewer occurrences yields the worst results (Feature Envy) and that participants tend to follow the advisor's suggestions. Regarding the best model, all of them performed relatively well; the differences were small, but in most cases the Random Forest model stood out. Regarding Crowdsmelling, the results show that the dataset evolves over time and is improving. The authors summarised the results as promising and are convinced of the feasibility of this approach.

(**3**) Ho et al. [36] suggested a novel method for detecting code smells using ML models. They introduced an approach called EnseSmells that combines statistical, semantic, and structural features of code. The approach consists of two main components, the Extractor and the Classifier. The extractor consists of two modules. The first mod-

3. Complex method implementing more than one functionality.
4. Class with large size, poor cohesion, with many responsibilities.
5. Method, which uses more functionality from other classes than from its own.

ule is the statistical semantic module, which compares code-language models such as CodeBERT, CuBERT, and code2vec to create source-code embeddings. To derive features from the embeddings, they use their DeepSmells model from previous work [37], which uses a Convolutional Neural Network (CNN). The second part is the structural module, which utilises CK [38] to extract 70 metrics from the code snippets. After preprocessing the metrics, they apply a deep neural network to effectively capture key structural features for code smell detection. The second component, the classifier, combines the vector outputs of both modules in the previous component using vector concatenation. The result is fed to the deep, imbalanced neural network for classification. Since the classes are imbalanced (more clean snippets than smelly ones), they introduced a sensitivity-weight adjustment to fine-tune the relative importance of the minority class. The results show improvements over existing state-of-the-art models of 6% to 28%, depending on the evaluated code smell.

Table 3.3: Related machine learning anti-pattern detection approaches

Topic	Techniques
Detection of code smells using relation-aware BERT and relation embeddings. Comparison with traditional ML techniques [32].	BERT models, relational embeddings, Gradient Boosting, Decision Trees
Experiment using the Crowdsmelling approach for gathering data for datasets to improve the dataset's reliability, focused on multiple code smells [35].	Crowdsmelling, Random Forests, Naive Bayes, J48, AdaBoostM1
EnseSmells approach combining statistical, semantic and structural features for detection of code smells. Uses about 70 metrics for detection [36].	BERT-based models, code2vec, CNN, DNN
WekaNose – machine learning based tool for detection of code smells. Allows dataset creation and ML models training [39].	Labeling, Sampling, ML algorithms
Approach for improving ML models performance by detecting code smells using continuous feedback. Results show continuous improvement of the model's reliability [40].	Naive Bayes, Random Forests, Logistic Regression, Code Metrics

4 JEAD: Java Enterprise Anti-patterns Detector

JEAD, formerly known as JENA [3][4], is a static analysis tool for detecting anti-patterns in Java enterprise development. The detectors are based on a catalogue of anti-patterns (see Section 3.1). This chapter introduces new features and improvements to detection. First, to provide more context, it focuses on the capabilities of the previous version. Then it describes the thesis's primary goal: introducing the possibility of detecting anti-patterns using machine learning models. It includes a description of the architecture, configuration options, and communication with the inference server that hosts the models (see Chapter 5). The last part discusses additional work done besides the primary goal.

4.1 Previous Version – JENA.V2

JENA.V2 introduced by Mička [4] is a command line interface tool based on the Spring Shell project [25]. The application can be controlled by a predefined set of well-documented commands and their parameters, which can be entered into the interactive CLI. It allows analysis of a project in 4 stages:

(**1**) **Project preparation** is executed using the command *prepareProjects*. As a parameter, it takes the target directory with the analysed project. If the project uses the Maven build system, it looks for *pom.xml* and adds the *maven-dependency-plugin* to expose JAR dependencies, which are then scanned by the Java Symbol Solver [27]. In Gradle, it appends a script to the *build.gradle* configuration doing the similar thing. The user then has to manually run the project's build to apply those plugins.

(**2**) **Configuration** is an optional step. Using the *copyConfig* command allows exporting the pre-defined configuration to a target path. This configuration can be manually adjusted for the specific project and used as input to the next stages.

(**3**) **The parsing and analysis** stage is executed using the *detectIssues* command. Arguments include the project path, the project label, the configuration path, and other settings. It uses the Java Parser and the Java Symbol solver to asynchronously parse the code. The code is parsed into the AST and each class (represented as *ClassOrInter-*

faceDeclaration) is passed to the detector. Each detector is associated with one *IssueCategory* (corresponding to the categories in Table 3.1) and can detect one or more issues in that particular category. The found issues are then automatically printed as a table in CLI and saved to database.

(**4**) **Presentation of results** stage can print the results using *aggregateIssues* command. It takes a project label as an argument and prints a table mapping issue categories to identify anti-patterns. The data can also be exported in the CSV format using *exportData* command.

4.1.1 Space for improvements

The current implementation of JENA.V2 opens a space for multiple improvements, which can enhance the reliability and precision of the detection and improve the user experience:

- JENA.V2 currently relies on rule-based detectors. The primary goal of this thesis is to analyse the possibility of using detectors based on ML techniques. It could improve the adaptability of detectors across different code bases without requiring adjustments to predefined thresholds.
- Support for the Lombok [5] library. JENA.V2 does not support annotation processors. Since detection sometimes depends on the presence of components in the code, this can lead to many undetected issues.
- JENA.V2 lacks a deployment package. At the moment, when a user wants to run JENA, they have to build the project from scratch or run it in an IDE. This really reduces the user experience when using the tool.
- Since JENA.V2 was developed almost 2 years ago, many dependencies are outdated. Also, the dependency structure in Maven's *pom.xml* is incorrect. Projects using Spring Boot [26] do not share a common dependency management approach, which may lead to incompatibilities.
- The project is composed of multiple sub-projects. They do not share the same naming conventions and are not correctly defined

as Maven modules. It is also unclear which sub-projects provide important functionality and which are just test fixtures.

4.2 Machine Learning Detectors

The main objective of this thesis is to improve JEAD with the detectors using ML models and to analyse its reliability and usability in terms of enterprise anti-patterns. The current version relies only on static rules defined in the configuration. The main problem with static rules is their inability to capture the semantic and structural features of code. They rely solely on a static threshold for a particular code metric (or regular expression, ...), saying whether this code snippet is an anti-pattern or not. Machine learning models can eliminate those problems, but they may involve longer analysis times and greater effort to build (see Section 3.2).

This chapter outlines the architecture of machine learning detectors, covering everything from configuration to code snippet extraction, and ultimately addressing communication with the dedicated server (see Chapter 5) that facilitates the deployment of machine learning models. Figure 4.1 presents the simplified detection flow for machine learning detectors. More details about each component are provided in the system-level diagram of the JEAD components shown in Figure D.1.

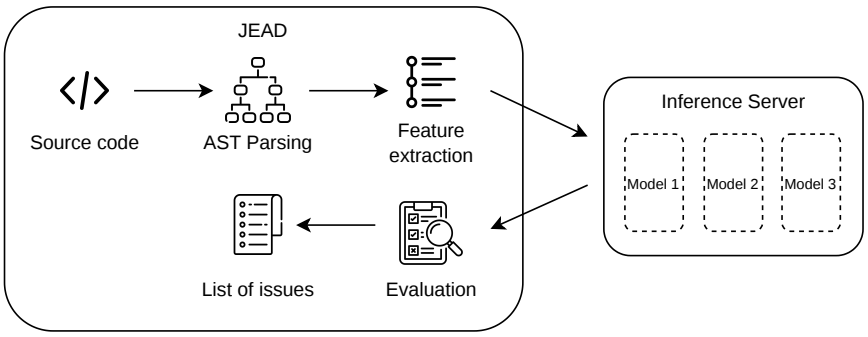


Figure 4.1: JEAD Machine Learning Detection Flow

4.2.1 Architecture

The architecture requirements primarily include independent parsing of code snippets and asynchronous queues for each model, so that calls to the Inference Server do not block parsing or processing of other code. Table 4.1 summarises basic architecture concerns related to the requirements.

The ML detectors work alongside the rule-based detectors, but they have different source root callback functions that are passed to the Java Parser and called for each parsed compilation unit. The ML callback is located in the *MachineLearningDetectorCallback*. Once the *detectIssues* command is executed, it loads the configuration for ML models (see Section 4.2.5) and delegates responsibility to *MachineLearningDetectorCombiner*. Sends each parsed Java class to the code extractors (see Section 4.2.2). Once the code extractor returns the model with extracted code features, it passes them to the *InferenceQueue* (see Section 4.2.3), which asynchronously sends the code features to the Inference server for inference. When the response with labelled snippets is returned, the results for each model are stored in its inference queue. When all code is processed, the handler for the *detectIssues* command stops all inference queues and synchronously waits for all found issues. The issues are then processed and merged with the issues from rule-based detectors, which are running in parallel. The simplified schema of the ML issue detection process and architecture is shown in Figure D.2.

4.2.2 Code Extractors

Since the ML models are trained on code features (code comments, code metrics, ...), not on entire code snippets, there must be a mechanism for extracting those features from the code. For this purpose, we introduced code extractors. Since the project uses the Spring Framework [41], the code extractors are implemented as Java classes registered as framework component beans. All code extractors must implement interface `CodeExtractor<T>`, which defines methods `Stream<T>extract(...)` and `String getIdentifier()`. The generic type parameter `<T>` defines the model class, which is returned by the extract method. Each model must implement the *EvaluationModel* interface

Table 4.1: JEAD architecture concerns description

Concern	Description
Interoperability	Uses gRPC interface to communicate with Inference Server.
Configurability	Settings for each model can be configured independently. The parameters for queues and evaluation are configurable.
Modularity	Code is easily extensible (through predefined interfaces) and easy to implement new functionality.
Performance	Handles more models simultaneously. Each model has a separate asynchronous queue.
Timeliness	Might introduce processing delay due to batch processing of items in the queue.

to identify the code snippet's position in the code during evaluation. The extract method is called by the *MachineLearningDetectorCombiner* with AST of the parsed class. The method processes the provided class and returns a stream of models, since a single class may include multiple code features. The *getIdentifier* method returns a string, which defines a unique identifier of a particular extractor used in the configuration to map the extractor to a concrete model name and detector (see 4.2.5).

It is also important to be able to build a training dataset for ML models. We introduced the *extractCode* CLI command, which allows extracting code features from a specified project using code extractors. The command also allows specifying the output type, with the user able to choose between JSON and JSON Lines (JSONL). The set of output formats can be easily extended via the *OutputFormatter* interface. The parameters of the commands are as follows:

- **configPath** (**c**): optional, path to the configuration file with static rules,
- **projectPath** (**p**): path to the target project,
- **format** (**f**): name of the output formatter,
- **extractor** (**e**): name of the code extractor,

- **outputPath** (**o**): path to the output file,
- **settings** (**s**): optional, additional options for code extractors.

Listing 4.1: Example usage of the extractCode command

```
extractCode -p /home/martin/source-project
            -f jsonl
            -e god-metrics
            -o /home/martin/god-di-metrics.jsonl
```

4.2.3 Inference Queues

Each configured ML model has its own inference queue. Their purpose is to asynchronously process extracted code snippets and send them to the inference server in batches. Such an approach significantly improves performance by creating an asynchronous gap between code extraction and inference. Since inference might be performance-intensive and time-consuming, a synchronous approach would block extraction until the item is processed. It would also create a network overhead, because each extracted item would have to be sent separately.

Inference queues are managed by the *InferenceQueueHolder*, which is responsible for the start and termination. Each queue has a configurable batch size and batch period (in milliseconds). The queue itself uses *ScheduledExecutorService* with a fixed rate set to the batch period. This means that the batch is processed every X milliseconds as defined in the configuration. Executor service uses virtual threads to improve performance and minimise platform thread overhead [42].

Once the code extractor processes the code and returns the evaluation model, it is wrapped in the *InferenceItem* record, which provides a unique ID and a mapping callback that maps the item evaluated by the ML model to the *Issue* entity. This *InferenceItem* is then passed to the *InferenceQueueHolder*, which selects the appropriate *InferenceQueue* according to the ML model name for scheduling. Once a thread in the queue is awakened at the batch period, it checks whether the queue has enough items (the batch size). If so, it empties the queue and forwards the batch to the inference server (see Section 4.2.4).

When the code extractors are finished, the *InferenceQueueHolder* terminates all queues. The termination sequence shuts down the executor service and flushes the queue in a synchronous loop, waiting until the queue is empty and all items are processed. In this state, the batch size parameter is not considered because the queue might not be full.

4.2.4 Communication with Inference Server

Since all ML models trained as part of this thesis were done using the Python language, it is necessary to deploy them somewhere. For this purpose, we introduce the Inference server, which, thanks to its modular architecture, enables efficient model deployment (see Chapter 5). The server provides gRPC API [43] through which the JEAD sends extracted code features for inference. The inference server returns either probabilities or labels as numeric values (depending on the selected model). The probabilities are then evaluated against the thresholds defined in the YAML configuration and converted into issues.

JEAD uses dependencies `spring-grpc-spring-boot-starter` [44] to autoconfigure the needed beans and `protobuf-maven-plugin` [45] to generate a client stub from the protocol buffer definition. The plugin then generates *InferenceServiceGrpc* service, whose blocking stub is registered as a bean in *GrpcClientConfig* configuration class.

The main communication logic is placed in the *InferenceService* class. The method *doInference* takes as a parameter a batch of items for inference and the name of the model. First, to create an anti-corruption layer, it maps the internal models to the gRPC models[1] using *InferenceMapper*. The resulting gRPC model includes just a list of evaluation items (ID and content, which represent the extracted code features). Then, it sends the blocking request to the inference server. When the response arrives, it maps the response payload back to the original internal *InferenceItem* model, including all additional metadata that are not sent over gRPC. This is possible thanks to a unique ID added to each evaluation item.

1. This ensures the gRPC models are not used internally in the JEAD. In case of API change, only the mapper is adjusted.

Listing 4.2: Example of model configuration in the application.yaml

```
inference:
  models:
    - &comments-model
      model-name: comments-model
      batch-size: 30
      batch-period: 100
```

4.2.5 YAML Configuration

The YAML configuration allows configuring the connection URL to the Inference server, specifying the ML models the JEAD should use for inference, and pairing the models with ML detectors that evaluate labels against predictions returned by the inference server. The configuration is located in `detector/src/main/resources/application.yml` or, in a ZIP bundle, in `config/application.yml`, together with other Spring Boot [26] properties. The application needs to be restarted after each configuration change. Here are the examples of the properties related to the ML and inference can be configured:

(1) **Models configuration** object includes defining `model-name`, `batch-size` and `batch-period` (those properties are mentioned in Section 4.2.3). The model's name serves to pair the model with the one deployed on the Inference server (unique identifier). The object must also have a defined anchor so that it can be referenced later. Listing 4.2 shows an example of a configuration for one model.

(2) **Detector configuration** first includes the `detector-name` property, but it does not have any special function; it is just for better readability of the configuration. Then, property `extractor` refers to a previously defined model. The extractor's name must match the identifier of a source code extractor (see Section 4.2.2), which will be used for extracting the code features sent to the Inference server. The `model` property must include an alias of the anchor defined in the `inference.models` object (an alias always starts with `*` symbol). The last property is `evaluations` which is described in point 3.

(3) **Evaluations configuration** defines the labels and thresholds that are returned from the Inference server. For each `label-name`, they

Listing 4.3: Example of ML detector and evaluation configuration in the application.yaml

```
inference:
  detectors:
  - detector-name: technical-debt
    extractor: comments
    model: *comments-model
    evaluations:
      - label-name: "technical_debt"
        threshold: 0.8
        issue-type: SELF_ADMITTED_TECHNICAL_DEBT
      - label-name: "code_comment"
        threshold: 0.8
        issue-type: COMMENTED_OUT_CODE
```

define `threshold` property. When the returned prediction is equal to or greater than the prediction threshold, it produces the issue with the defined issue type in `issue-type` property. The issue type must be one of the values defined in *IssueType* enum. Listing 4.3 shows an example of an evaluation configuration.

(4) Inference Server configuration allows the configuration of the URL to the gRPC interface of the running Inference server instance in the `address` property. Then, the `enabled` property allows us to define whether the ML inference and detectors should be used. If disabled, only rule-based detectors are used. See Listing 4.4 for an example.

4.3 Additional Work

The following section introduces support for a tool called Lombok [5], which was not supported in the previous thesis and limited JEAD's usage in the newer projects. The following section then continues with a description of an enhanced project structure to simplify future development, as well as the creation of a deployment package that

Listing 4.4: Example of Inference server configuration in the application.yaml

```
spring:
  grpc:
    client:
      channels:
        inference-server:
          address: localhost:8081

inference:
  enabled: true
```

bundles the entire application and releases it as a self-contained ZIP file.

4.3.1 Lombok Support

The Lombok [5] library provides a set of annotations to omit boilerplate code. When the annotation is used (for example, *@NoArgsConstructor*), it uses the annotation processor to generate the code for a no-argument constructor in the class where the annotation is used. Another example is annotations such as *@Getter* or *@Setter*, which generate getters and setters for fields in the particular class. This generated code is not visible in the *.java* source files that are used by the Java Parser to compose AST. This lack of actual source code in the *.java* files prevents the Java Parser/Java Symbol Solver from correctly analysing the source code, resulting in imprecise metrics for evaluating anti-patterns.

For this purpose, the binary (*.jar* file) of the Lombok dependency includes **Delombok** [46]. It is a primitive CLI application that takes a directory containing source code and a path to the output directory as input. Then it generates Java source code that resolves all Lombok annotations, enabling full static analysis.

For this purpose, we introduced JEAD's Maven and Gradle plugins, which can be included in the analysed project. These plugins are included using the *prepareProject* command in the build description

scripts (*pom.xml* or *build.gradle*). During the project build, the plugin delomboks the code and creates a separate folder containing the resolved code. We also moved the functionality of `maven-dependency-plugin`, introduced in the previous version [4], to the newly introduced plugins. Its purpose is to expose all *.jar* dependencies so the Java Symbol Solver can correctly resolve all types of third-party libraries. This eliminates the need for multiple included build plugins in the target projects.

The *prepareProject* command modifies the build configurations and includes build plugins as defined in application.yaml. There is a preconfigured, correct version of the JEAD plugins that matches the current version of the application bundle (see Section 4.3.4). The versions are resolved at build time using `maven-resources-plugin` [47], which replaces placeholders with the concrete value defined in the *pom.xml* by the automated GitHub Workflow. The workflow also uploads the plugins to GitHub Packages [48], so the plugins can be automatically downloaded and do not need to be added manually. Both plugins use the `jead-plugin-core` library, which includes executors for both tasks/goals. The project structure overview is shown in the Appendix B. Both plugins are built with Java 11 compatibility, so they can also be used in older projects.

Listing 4.5: Example usage of the prepareProjects command

```
prepareProjects -d /home/martin/analysed-project
```

JEAD Maven Plugin

The Maven plugin is included in the `<build/>` tag of *pom.xml* in the target project. It includes two goals: `delombok`, `copy-dependencies`, which are by default bound to the `package` phase. Both goals include configurable parameters that can be used to customise behaviour, but the default configuration should be sufficient in most cases. The following steps should be applied to successfully prepare a Maven project for analysis:

1. Execute prepareProjects command (see Listing 4.5).
2. Run `mvn clean package` in the target project.

3. Check folder `target/dependencies` for exposed *.jar* files and `src-delombok` for resolved Lombok source code (if Lombok is used).

JEAD Gradle Plugin

The *prepareProjects* command includes generic build script blocks in the *build.gradle*. However, manual adjustments may be required, since the correct position of the included script may differ between projects, and cannot be determined automatically. Similar to the Maven plugin, it includes two tasks: *jeadCopyDependencies* and *jeadDelombok*. But none of them is bound to any other Gradle task, so they have to be executed manually. The following steps should be applied to successfully prepare a Gradle project for analysis:

1. Execute prepareProjects command (see Listing 4.5).
2. Run `gradle build` in the target projects.
3. In case of some errors, manually fix the *build.gradle* and run again.
4. Run `gradle jeadCopyDependencies` to expose *jar files*.
5. Run `gradle jeadDelombok` (only if the Lombok is used).
6. Check folder `target/dependencies` for exposed *.jar* files and `src-delombok` for resolved Lombok source code (if Lombok is used).

4.3.2 Dependency Updates & Migration to Java 25

Since JENA.V2 [4] was developed almost 2 years ago, it included multiple outdated third-party dependencies. Also, we updated from Java 17 to Java 25 to support the latest language features. During the update, we also fixed and improved the structure of the *pom.xml* files. Some dependencies, like Spring Boot, were not handled correctly. There was a missing `spring-boot-dependencies` entry in *dependencyManagement* section, which enables correct automatic versioning of dependencies under the Spring Boot framework [26]. We also had to

Table 4.2: List of selected updated dependencies

Group ID	Artifact	Old v.	New v.
org.springframework.boot	spring-boot-starter	3.1.1	3.5.10
org.mockito	mockito-core[a]	5.2.0	5.21.0
org.postgresql	postgresql	42.6.0	42.7.9
net.bytebuddy	byte-buddy[b]	–	1.18.4
org.hibernate	hibernate-core[c]	6.2.7	6.6.41
tools.jackson.core	jackson-databind	2.15.2	3.0.4

a. Changed from mockito-inline to mockito-core.
b. Added to preserve mocking support.
c. Removed explicit dependency. By default included in Spring Boot.

update the Maven and Gradle wrappers to their latest versions due to incompatibility with Java 25.

The following table lists some selected updated dependencies. Not all dependencies were updated to their latest versions due to instability or incompatibilities. One of the examples is the Spring Boot framework. At the time the thesis was being worked on, version 4 had just been released. Some dependencies, like Spring Shell [25] included some bugs and major changes preventing the smooth use of the application.

4.3.3 Improved Project Structure

The structure of JENA.V2's sub-projects was not ideal and was harder to understand. This thesis slightly improved the structure by ensuring naming consistency, a correct Maven project structure, and automatic build of Gradle projects via Maven. The visualisation of the new structure is in the Appendix B.

The JEAD includes several test fixture projects[2]. They allow running unit tests for rule-based detectors against real projects. We moved them into a subproject called `test-fixtures`, which makes the structure clearer and separates non-production projects.

The new build plugins are then located inside `build-plugins` subproject. They share common Maven properties and a few dependencies

2. *antipatterns*, *authorization-server*, *power-mock-usage* and *test-gradle-project*

that are not relevant to the `detector`. The overall structure of Maven modules was improved, so it is possible to build or install all projects from the root folder using `mvn clean install`. Since two projects use Gradle [3], we also added to both projects Maven's *pom.xml* with logic that executes Gradle tasks specific to each Maven phase.

4.3.4 Enhanced Deployment

To provide users with a better experience, we decided to create a package that includes the built *jar* application, the configuration files, Docker Compose files required to run the Inference Server (see Section 5.7), and a comprehensive README file with all the necessary instruction for running the application. The bundle is published as a compressed *zip* file. The user can just download this bundle, unpack it, and run the prepared *jar* archive directly, or start the Docker containers without building the whole application. The creation of the bundle is done by the *maven-assembly-plugin* [49], with the descriptor located in the Maven sub-project `assembly` (see the project structure in Appendix B). The structure of the *zip* package is shown on Figure 4.2.

This runtime assembly package is automatically built using GitHub Actions [50]. The action is triggered when a Git tag is created whose name starts with the letter "v", for example, v1.0.1. The action performs the following steps to build and package the application:

1. Extract the version from the Git tag and set it for Maven projects.
2. Run tests and install the JEAD Maven and Gradle plugins.
3. Publish JEAD Maven and Gradle plugin artefacts to GitHub Packages [48].
4. Run tests for JEAD detector.
5. Package the JEAD detector into the *zip* archive.
6. Create a GitHub Release with the version and upload the *zip* archive to it.

---

3. *test-gradle-project* and *jead-gradle-plugin*

```
JEAD/ ..................................... Root directory
├── config/
│   ├── application.yml ................... Spring Boot config
│   └── configuration.json ................ Detection config
├── .env ................... Environment variables for Docker
├── compose.gpu.yaml ............ GPU-compatible Compose file
├── compose.yaml ............... Compose file without GPU comp
├── detector.jar ............. JAR archive with JEAD DETECTOR
└── README.md ..................... README with instructions
```

Figure 4.2: Structure of the JEAD zip package

Since the detector requires a database and optionally an Inference Server, Docker [51] and Docker Compose [52] must be installed, or an equivalent alternative must be used. The Appendix C describes the necessary steps to run the detector and evaluation. Those steps are also included in the `README.md` file in the JEAD runtime package.

5 Inference Server

This chapter describes the purpose of the Inference server and its technology background. First, it summarises existing solutions and why we have decided to create a custom one, then it describes the architecture and the technologies used. The following part discusses the project structure, the public interface, and, ultimately, the Dockerization and deployment of the server, including Graphics Processing Unit (GPU) compatibility.

The purpose of the Inference server is to enable the deployment of ML models via a unified API. Since most AI development is done in Python, mainly due to its simplicity and strong library support. The JEAD detector is written in Java, and it would not be possible to call Python functions easily and reliably. Due to these facts, we decided to create Inference server, a modular gRPC server that allows hosting ML models in separate subprocesses with independent access to GPU cards, deployable in Docker [51] containers. The ML models are delivered outside of a Docker image in a pre-defined file-based structure called "models root". This allows for small, modular Docker images. A user can deploy the model by adding a folder containing a defined Python worker file. Then, after proper configuration, the JEAD sends the extracted code snippets to the Inference server. The server forwards snippets to the specified model and returns the response with predictions, which are then further processed by the JEAD.

5.1 Existing Solutions

There are multiple robust and ready-to-use solutions. But most of the solutions were too complicated for our simple use case. The requirements for the Inference server are:

- Small Docker Image,
- modular API, preferrably using gRPC,
- modular (plugin-based) architecture that allows adding models without server code modification,

Table 5.1: Existing Inference Server solutions

Name	FOS	Runtimes[1]	REST	gRPC	Image Size[2]
NVIDIA Triton Inference Server [53]	✓	✓	✓	✓	– / > 4 GB
TensorFlow Serving [53]	✓	✗	✓	✓	200 MB / 2 GB
MLServer [54]	✓	✓	✓	✓	– / > 4 GB
KServe [55]	✓	✓	✓	✓	– / > 250 MB

- free and open source,
- can host models built by different Python libraries,
- allows running easily on the user's machine.

Probably all solutions presented in Table 5.1 could be used to infer JEAD's code snippets, but are often not well-suited for deployment on a local machine or produce very large Docker images. We decided to build our own solution tailored to our simpler use case. The solution can also be used across different use cases and integrated with various applications. Some of the concepts were inspired by NVIDIA Triton Inference Server [53]. Custom solution also allows us to customise the API and use cases according to our needs.

5.2 Architecture

As a primary structure of the project, we decided to use a layered architecture. It brings the benefits of simplicity, low development effort and separation of concerns. Since the project is relatively small at the moment, we use 3 layers. In a typical application, the bottom layer is the database layer. But since our Inference server does not communicate with the database, but communicates with ML models,

2. Support of more Python libraries/runtimes for model definition.
2. CPU / GPU compatible image size. We always show the minimum size found; it often depends on the selected runtime.

it is a bit special and differs from the standards. The layers and their responsibilities are:

1. **Presentation Layer** – gRPC interface (see Section 5.4),
2. **Business Layer** – business logic and validation, models storage, loading, and unloading ML models,
3. **Models layer** – communication with deployed models (see Section 5.6).

The architecture leverages Dependency Injection principles [56]. We do not use any framework; we manually create services during application startup and manually inject dependencies into the constructors of dependent objects. This way, we ensure testability by unit tests of all dependencies, since they can be easily mocked. Thanks to this, we implemented a set of unit and integration tests to ensure the server's reliability.

The application is written in Python language. To improve performance, we use an asynchronous programming model, primarily using the *asyncio* library [57]. Thanks to this, Python can handle multiple tasks in the event loop simultaneously, significantly improving throughput. However, the main bottleneck of the Inference server is always the ML model, because models often require substantial resources to provide fast inference, including GPU resources. The *asyncio* library is also widely used for communication with ML models, where Futures enable handling multiple commands in parallel. Without that, only one command could have been processed at the same time (more on this in Section 5.6).

The project and its dependencies are managed by the Poetry [58]. It allows us to define *pyproject.toml* file with a list of all dependencies, together with their versions and other metadata. After installation, the produced lock file can be committed to Version Control System (VSC) to ensure that all dependencies have the same versions across different environments.

```
inference-server/
├── proto/ ..................... Protobuf definitions folder
├── .env.example .... Template for of environment variables[a]
├── compose.local.yaml ..... Compose file for local run[b]
├── compose.yaml ............... Compose file for local run[c]
├── Dockerfile ........ Docker container build specification
├── poetry.lock ........................... Poetry lock file
├── poetry.toml ............... Poetry project specification
└── REAEDME.md ............... README file with instructions
```

a. Should be copied and renamed to .env file before starting the server locally.
b. Builds the project using Dockerfile
c. Fetches the latest Inference Server from DockerHub.

Figure 5.1: Structure of the Inference Server project

5.3 Project Structure

Figure 5.1 provides a general overview of the project's structure, without details about the source code structure, which is omitted for simplicity. The source code folder follows a package-by-layer structure.

5.4 Public gRPC Interface

The Inference server uses the gRPC [43] interface on the presentation layer. gRPC was chosen mainly for its ability to provide typesafe clients and stubs, as well as its lightweight binary format. On the server side, we use the Python library *grpcio* [59]. The server stub is generated from the protobuf definition in the *proto* folder at the project root. To generate, the script *scripts/generate-server.sh* must be run after installing the Poetry project. The interface now includes 5 endpoints located in the service called *InfereceService*:

- **ServerReady** for checking whether the server is online,
- **ModelReady** for checking whether the model is ready for inference and loaded,
- **LoadModel** to load the specific model,

- **UnloadModel** to unload the specific model,
- **ModelInference** to run the inference on the specific model.

Currently, only the *ModelInference* and *ServerReady* are used from the JEAD detector. The first one mentioned uses the *InferenceRequest* message as the payload. The Listing 5.1 shows the protobuf description of that message. As the first parameter, it takes the model's unique name for which inference should be performed. The second parameter is the list of *InferenceRequestContent* messages. It includes the unique ID of that message (the ID is also returned in the response, so JEAD can pair the requested code snippet with the inference result). The content includes a string, which is then passed directly to the model's worker. The string typically contains JSON, which the model can parse. The use of the JSON format is a trade-off between the truly dynamic structure of the API endpoint and lower performance due to parsing, with more overhead than the pure binary format gRPC provides. However, since each model requires different data in different structures, this seemed to be the optimal solution.

Listing 5.1: InferenceRequest message protobuf definition

```
message InferenceRequest {

  message InferenceRequestContent {
    string id = 1;
    string content = 2;
  }

  string model_name = 1;
  repeated InferenceRequestContent contents = 2;
}
```

5.5 Models Root Folder Structure

To have a truly dynamic modular/plugin architecture, each model is basically a separate plugin. This plugin structure consists of a set of folders, each containing a specific Python script. The root folder

```
models_root/
├── comments_model/
│   ├── worker.py
│   └── model/
│       └── model.onnx
├── multi_service_model/
│   ├── worker.py
│   └── model/
│       └── model.onnx
├── requirements.txt
└── requirements.gpu.txt
```

Figure 5.2: Example of models root folder structure

is called "models root". It must include two files: *requirements.txt* and *requirements.gpu.txt*. Both files include Python dependencies in the Requirements File Format [60]. Those dependency files should include dependencies of all ML models the user wants to deploy. The primary motivation is to split the dependencies of the Inference server itself and those of ML models so they do not conflict. Then, the model's root contains ML models, each in a separate folder. The name of the folder serves as a unique identifier (name) of the model. Each folder must include a file named *worker.py* containing a class named *MLWorker* with predefined methods (see Listing 5.2). Then, each folder can contain files and folders specific to each ML model, which are handled by the user by logic in the *MLWorker* class. The Figure 5.2 shows an example of the models root folder structure.

Listing 5.2: MLWorker class template

```
class MLWorker:
    def load(self):
        ...

    def unload(self):
        ...

    def execute(self, data):
        ...
```

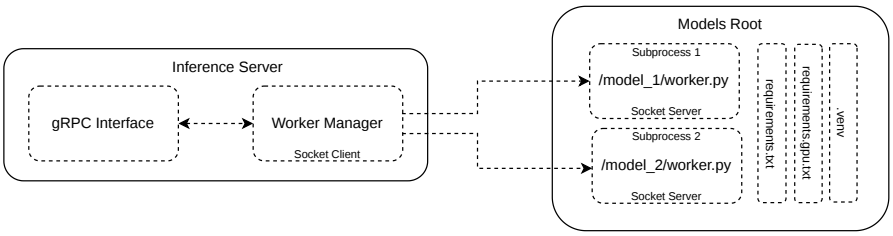


Figure 5.3: Simplified architecture of Inference Server with models root folder

5.6 Models Loading and Communication

During the server start-up, it checks for the requirements file. Depending on the mode (GPU or CPU), the correct requirements file is chosen. Then, it creates a Python virtual environment and installs the dependencies using pip [61] within it. When the model is loaded, it is started as a separate process within this environment. The process is started using Python's internal *subprocess* library. It starts the *socket_worker.py* script, which is shared for all models. This script acts as a socket server, started on a randomly picked port. The Inference server itself is then a socket client that sends commands over the socket interface to the ML models[3]. The Inference server manages connections and processes. Each sent command is wrapped with a request ID, and all pending requests are stored in a dictionary together with *Future* (from *asyncio* library [57]), so a response can be processed asynchronously. This also allows processing more requests simultaneously over a single socket stream.

Since the socket server script is shared across all models, there must be a way to dynamically load the *worker.py* script for the concrete model. To achieve this, we utilized *importlib* package from Python's standard library. It provides utilities to load a Python script on a defined path as a module. Once the module is loaded, the script has access to the defined *MLWorker* class, which can be easily instantiated and its methods called.

Since the internal socket server script also runs in the virtual environment, where a user has defined their own dependencies, all code

3. There are 3 commands available: inference, load, shutdown.

should use only modules from the standard Python library to avoid conflicts with those dependencies.

The diagram in Figure D.3 shows the server's simplified modular architecture. It includes examples of two deployed ML models.

5.7 Dockerization and Deployment

The application is containerised as a standalone without ML models. This allows the creation of small Docker containers. The production container is automatically created using GitHub Actions [50]. The action is triggered when a Git version tag, starting with the letter "v", is committed, for example, v1.0.1. The action first installs the Poetry project and then generates the gRPC server stub (see Section 5.4). Then, it runs unit and integration tests. In the end, it builds a Docker image and publishes it to the Docker Hub repository [62].

The JEAD *zip* package (mentioned in Section 4.3.4) includes a compose file that downloads the image from a remote repository and runs a local container. There are two compose files: *compose.yaml* and *compose.gpu.yaml*. The second is just an extension of the first, which allocates NVIDIA [63] graphics cards to the running container. This means that the GPU compose file cannot be run separately. Before starting the container, the *zip* package includes an *.env* file with two properties. The property *MODELS_ROOT_HOST* defines a path to the model root directory. The property *USE_GPU* is a boolean that determines whether the Inference server should be started in GPU-compatible mode and install the model's GPU dependencies. The compose file in the *zip* package always includes a tag for the correct version of the Inference server compatible with the compiled JEAD detector.

The model root folder must be located on the host machine. It is then mapped to the container as a volume. When Inference Server creates a virtual environment, it is created inside the model's root folder and populated with the installed dependencies. Note that some dependencies are compiled specifically for a concrete device or operating system, so the virtual environment used when running the server outside a Docker container will most likely not work inside the container.

The Appendix C includes a guide on how to run Inference Server using Docker, including the JEAD detector and running the evaluation.

5.8 Future Work

This thesis aimed to create an Inference Server mainly for inference on local environments. Thus, it did not implement any security aspects. Future work should focus on securing the gRPC interface using an API key included in the request header. This API key would also be configured in JEAD, and only requests with the correct API key would be allowed. If needed, a more sophisticated authorisation mechanism could be considered, for example, permissions to access concrete models bound to a specific API key/user.

Another security aspect is encrypted communication. At the moment, the communication between JEAD and the Inference server is not encrypted. The same applies to TCP socket communication between JEAD and ML model workers. Both communication protocols should be secured and encrypted using TLS authentication.

The models are automatically deployed when the first request for the concrete model arrives. Or, they can be deployed using a separate endpoint. However, the purpose of deploying each model separately is to save resources. If a future requirement is to run the Inference server as a "permanent" service, it should include an automatic mechanism to undeploy unused models. The model could be undeployed if it is not used for a period of time.

More API endpoints could also improve the server's usability. For example, some endpoints listing available or deployed models might be helpful. With this, some standalone CLI application, or another type of User Interface (UI) could be created to provide more control over the Inference server. The UI can also be included in the JEAD detector CLI.

6 Detecting anti-patterns Using Machine Learning Models

This chapter describes the process of training ML models to detect enterprise Java anti-patterns. From the catalogue of anti-patterns (see Appendix A), we selected three anti-patterns suitable for ML detection. Each anti-pattern uses a bit of different detection technology and approach to demonstrate the possibilities. For each model, we discuss an approach to creating the training/fine-tuning data set, a data preprocessing pipeline, training the model, and converting the model to the ONNX format [64].

The ONNX format is a standard representation of ML models. It allows reducing the Python dependencies used for inference, together with a unified API for all models to obtain predictions. It also supports hardware acceleration (GPU). Table 6.1 shows an overview of trained anti-pattern detectors together with the technologies used.

All pipelines used for data preprocessing and training are in the Poetry *training* project. Each pipeline is created as a Jupyter Notebook, which requires manual execution of each needed block. Training or fine-tuning some models may require a GPU, especially transformer-based models. Some datasets were enhanced with LLM samples, which require an API key for access. Datasets are available in the thesis attachment.

6.1 Anti-patterns Selection Motivation

The catalogue of anti-patterns (see Appendix A) includes multiple anti-patterns, which are suitable for detection using ML models. The catalogue provides a brief analysis for each suitable anti-pattern. We selected three of them, each representing a different technique and technology used for detection. The purpose of those selected models is to create a proof of concept for ML detection, and to verify that the selected architecture works correctly.

The DI8 anti-pattern is a detection technique based on code metrics. The metrics are extracted from the code using JEAD and then passed to a metric-based model for evaluation. Another category, which in-

Table 6.1: Overview of ML anti-patterns detectors with technologies used

Anti-pattern	Technology	Python library
DI8 God DI	Random Forests model	scikit-learn [65]
SER1 Multi Service	CodeBERT for embeddings, HDBSCAN for clustering	Hugging Face tranformers [66], scikit-learn [65]
TECH_DEBT	CodeBERT for classification	Hugging Face transformers [66]

cludes the SER1 anti-pattern, involves using unsupervised learning techniques in which there is no dataset to learn model-specific patterns, but techniques like clustering are used instead. The last category presented uses a supervised learning approach, based on the semantic analysis of the code (comments).

Those categories mentioned in the previous paragraph represent the most often used ML techniques. There are definitely more techniques and anti-patterns in which ML learning can improve detection capabilities. For example, a model can help detect security tokens stored in insecure locations (SEC1) rather than relying solely on regular expressions. In the case of the DI11 anti-pattern, a model could help identify computationally intensive services. Then, with the aid of static analysis, the locations where lazy initialisation should be added can be determined.

6.2 (DI8) God Dependency Injection Detection

This anti-pattern is purely metric-based. The rule-based detector implemented in JENA.V2 [4] relies on a number of injected fields evaluated against a static threshold. This static threshold often generalises poorly across projects and may need to be manually adjusted. To fully utilise the model's capabilities, we decided to increase the number of code metrics used for detection, since the number of injected fields alone

might not be representative enough. The metrics used for the detection are:

- **Cyclomatic Complexity** calculates the amount of decision logic to express code complexity,
- **Lack of Cohesion of Methods 5** measures how many methods share access to the common attributes,
- **Number of Object Methods** in the class,
- **Number of Class Methods** in the class,
- **Number of Injected Fields** injected by some Dependency Injection framework,
- **Lines of Code**.

There are many approaches to training models based on metrics. Fontana et al. [67][68] use Random Forests, Naive Bayes, J48 or JRip models. Azeem et al. [69] in a systematic literature review mention that Random Forests, together with J48, are selected as the most effective algorithms in multiple studies. We decided to use Random Forests, primarily for its good balance between performance and resource heaviness. It also has good support in the scikit-learn library [65]. We trained a Random Forest model for a binary classification problem, where *label 0* represents a clean code snippet and *label 1* represents a snippet containing an anti-pattern.

Dataset

One possibility to obtain the dataset is to use the JEAD code extractor to extract code metrics from various open-source projects. But this would include processing many open-source projects to get a representative number of code metrics. Since code-metrics datasets are relatively common, we used the open-source Classes and Metrics dataset (version 3.2.2024) [70]. It includes more than 48 metrics from 1,000 GitHub repositories with 532,000 Java classes.

The dataset uses a folder structure. Each project in this dataset has a folder containing CSV files. Each CSV file includes data for

a particular metric. Since this structure would be too complex for labelling, we created a custom script to extract those metrics into JSONL, where each JSON line represents a Java class and all required metrics.

To label the snippets, we used a combination of automatic labelling and manual inspection of samples. Automatic labelling was performed using a custom script, which defines "extreme" values for each metric. For example, if the number of object-oriented methods is greater than 50, the provided code snippet will most likely be an anti-pattern. This helped us reduce manual labelling effort, but of course, it could introduce some biases. To minimise bias as much as possible, we used a random sampling approach for manual inspection. Snippets that were not labelled by this automatic approach were labelled manually. To make working with JSONL and CSV files easier and more unified, we utilised the Pandas library [71].

Training

For training the Random Forests model, we use the scikit-learn library [65]. First, the prepared dataset is loaded into a DataFrame using the Pandas library. We then split the dataset randomly into training and test sets, allocating 20% to the test set. For the actual training, we use *RandomForestClassifier*, which handles the training automatically. Then, we convert the trained model to the ONNX standard to simplify deployment. For this purpose, we use a separate library called skl2onnx [72]. The output is a single file containing the model in the desired format.

6.3 (SER1) Multi-service Detection

The Multi-service detector is a bit different from the other two. To create this detector, we did not train any model; instead, we utilised existing algorithms and models. Mička [4] defines the SER1 Multi-service anti-pattern as a service with many unrelated methods using completely different types. The rule-based detector for this anti-pattern uses only the number of methods, compared against a static threshold, to determine whether it is present.

We took the concept of detecting this anti-pattern more from the developer's point of view than from a code-metrics perspective. Developers often detect this class by analysing the semantic meaning of method names and the types methods take as arguments or return. When a class has many semantically unrelated methods, it is often marked as an anti-pattern[1].

Sivakumar et al. [73] used CodeBERT and CodeT5+ transformer models to generate embeddings. After post-processing in the dimensional reduction algorithm and clustering, they are able to capture semantic similarity and visualise similar solutions in the software projects. To perform a semantic analysis of the method, we extract method signatures from the tested classes using the JEAD extractor, generate token embeddings and apply a clustering algorithm to find the number of clusters. The number of clusters can indicate whether there are any semantically unrelated groups of methods.

The ML detector uses CodeBERT [33] to generate high-dimensional token embeddings from method signatures. Each method signature is represented as an array of vectors (one vector per token). To obtain a vector representation for each sentence, we apply the mean pooling technique. Most clustering algorithms do not work well with high-dimensional data (CodeBERT embeddings have 768 dimensions), so we perform dimensionality reduction using UMAP [74]. The resulting vector includes about 50 dimensions, while preserving most of the semantic-related vectors. In the end, we use the Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [75] algorithm to perform clustering. It is an extension of the DBSCAN [76] algorithm that allows finding more accurate clusters even when each cluster has a different density. The advantage of both algorithms is that they do not require knowledge of the expected number of clusters, unlike algorithms like K-Means.

After we pass dimensionally-reduced embeddings to HDBSCAN, we get an array, where each item represents a cluster number or noise. With that, we can compute the total number of clusters while ignoring the noise for our application. Then, if the number of clusters is

1. For example, the class CarService includes methods *createCar*, *updateCar*, *removeCar*, but also unrelated methods like *startRace*, *endRace*, *getRaceTime*, which should be most likely moved to RaceService.

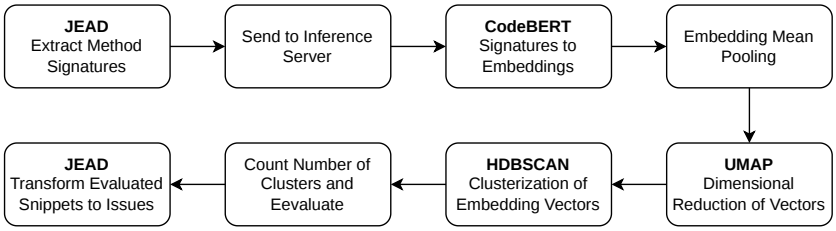


Figure 6.1: Multi-service detector machine learning pipeline

greater than 2, we mark the class as an anti-pattern. This threshold was deducted and fine-tuned during evaluation on multiple projects. The threshold 1 seemed too strict, and the pipeline consistently identified more false positives. We also ignore classes with fewer than 10 methods. In such cases, the clustering algorithm lacks sufficient samples to correctly identify clusters. Figure 6.1 shows a visualisation of the whole pipeline.

6.4 (TECH_DEBT) Technical Debt Detection

The Technical Debt detector is the only of the three ML detectors that is not implemented as a rule-based detector, because this anti-pattern was added to the catalogue in this thesis and thus not implemented with other rule-based detectors in JENA.V2.

Technical debt refers to suboptimal solutions deliberately introduced into codebases, typically when there is time pressure to deliver functionality, and the team agrees to implement workarounds. There are many types of technical debt, but we focus only on Commented-out Code and Self Admitted Technical Debt (SATD), which indicate that some functionality requires future attention. Li et al. [77] published an approach leveraging a BERT-based deep learning approach to detect SATD. They used code comments, commit messages, pull request comments, and issues to create a dataset for training a BERT-based model. Mahmood [78] introduced an approach using CodeBERT [33] to detect various types of inline code comment smells, including commented-out code, and compared the results with other ML techniques. The CodeBERT model significantly outperforms other techniques.

We decided to fine-tune one CodeBERT model for both tasks using a multi-label classification approach.

Dataset

We used the JEAD code extractor to manually extract comment codes from the open-source project. Table 6.2 presents all the projects that were used to create dataset. There are three types of Java comments – JavaDoc, Line comment and Block comment. The JavaDoc and Line comments are pre-processed in following ways.

(**1**) **JavaDoc** comments are separated into two parts: the description and tags. One separate comment snippet is created from the description. Another comment snippets are created for each JavaDoc tag. This helps reduce the length of separate comments, because JavaDoc comments always contain a lot of information. Also, the semantic meaning is often slightly different between the description and each tag.

(**2**) **Line** comments are merged. When multiple consecutive lines contain the Java Line comment, they are merged into a single comment. Line comments are often used instead of block comments because developers can select multiple lines and use a shortcut to create line comments. This results in the larger "block" of line comments. We squash comments because such blocks can often include semantically relevant information.

The dataset was then manually labelled. It includes about 3,000 samples per class. Clearly, most of the classes were for "clean" comment labels, so we used random undersampling to balance the data set. We also used LLMs models to generate artificial samples and variations from examples in open-source projects. 20% of the dataset is used for testing, and the remaining 80% for training.

Fine-tuning

The fine-tuning of the model is done using the HuggingFace [66] library. It includes a comprehensive set of utilities to easily download, fine-tune, and save ML models from their library. Before running the training script, we preprocessed each comment by adding a special token representing the comment type (JavaDoc, Block, or Line) at

Table 6.2: Projects used for technical debt model fine-tuning

Project Name	Commit Hash	Source
Apache Hive	d89f7c55	[79]
Apache Tomcat	c16254d	[80]
ThingsBoard	73a60d3	[81]
PA165 projects[a]	–	[82]

a. Set of 30 projects from the PA165 course tought on Masaryk University in Brno. Source code is not publicaly available.

the beginning. This way, the model can learn differences between individual types. For example, JavaDoc comments may include HTML or Markdown code, whereas other types cannot. The special tokens are also passed directly to the model as an argument. The Listing 6.1 shows an example of how comments should look when passing to the model. Since we fine-tuned the model with this format, it must also be used at inference time.

Listing 6.1: Example of comments with special tokens

```
[JAVADOC] JavaDoc Comment
[LINE] Line Comment
[BLOCK] Block Comment
```

After such preprocessing, each comment is tokenised using the related tokeniser. Each transformer-based model available on HuggingFace has its own tokeniser, which is included with the model, ensuring compatibility. The tokenised inputs are then passed to the Trainer API, which handles the training and all related operations. Training is carried out in three iterations (epochs). Fine-tuning is performance-intensive, so a GPU is highly recommended. Otherwise, the time required for fine-tuning can be long.

After training, the model is exported to the ONNX format [64]. During inference, the model needs a GPU card for faster evaluation. It can also run in Central Processing Unit (CPU) mode, but performance is significantly influenced. For inference, we use *onnxruntime* [83] library. After passing the comment snippets to the ONNX session, it returns

probabilities. They are then sent back to the JEAD, where evaluated against probability thresholds defined in the configuration. Based on this, JEAD creates issues if needed.

7 JEAD Evaluation on Open Source Projects

This chapter presents the results of the JEAD evaluation on open-source projects. The first chapter discusses the selection criteria for open source projects used for evaluation. The second part describes the steps for preparing the target projects, analysing them, and processing the results. The evaluation focuses mainly on machine learning models (see Section 6) integrated into the JEAD, compared to the rule-based detectors introduced in the JENA.V2 [4]. The last section discusses threats to the evaluation's validity, including limitations of the thesis, dataset size, and reproducibility.

The goal of the evaluation is to demonstrate that the selected architecture and machine learning models are comparable to rule-based detection. For evaluation, we only use the machine learning models we mentioned in 6.1. If the model has an equivalent rule-based detector, we show the comparison to demonstrate the differences and the ability to detect anti-patterns correctly. Other rule-based detectors are omitted, since this thesis introduced only minor changes and the results would not add additional value compared to the evaluations in previous works [3][4].

7.1 Project Selection Criteria

The main criterion for the selected project is its size, which should allow manual inspection of classes to manually label the reference dataset. Then, the JEAD issue-detection results will be evaluated against this ground truth. Thanks to that dataset, we will be able to build a confusion matrix, enabling us to calculate more detailed metrics for evaluating the model's performance. This approach is used only for models that detect class-level SER1 and DI8 anti-patterns. The ML model for TECH_DEBT would require manual examination of all comments in the respective codebase, potentially resulting in hundreds or thousands of samples. In contrast, the project should not be small, as anti-patterns like SER1 are more likely to emerge in larger codebases.

Projects are searched on GitHub. They have to be written in Java using the Spring Boot Framework. We are also searching for actively

maintained projects (with the latest commits not older than 1 month) that support at least JDK 17. This should, at least partially, ensure that the projects are up to date and use the latest practices in Java enterprise software development. Because JEAD is highly configurable, using the Spring Boot framework is not strictly required. However, its default configuration is tailored for Spring Boot, which is also currently the most commonly used web framework in the Java ecosystem. The last condition is that the selected projects should include at least some representatives from each anti-pattern mentioned in the evaluation section.

The selection of projects is relatively limited, likely due to the proprietary nature of enterprise-grade software. According to the previously mentioned criteria, we selected two projects:

- Klaw [84],
- Apollo [85].

7.2 Evaluation Strategy

As mentioned in the previous chapter, we manually labelled a ground-truth dataset for both projects to evaluate DI8 and SER1 models. This allows us to build a confusion matrix[1] [86] for both ML detectors and also rule-based detectors. The confusion matrix is used primarily in ML applications to evaluate classifiers, but since rule-based detectors are also classifiers, we can use it to compare the performance of both approaches. Comparing both approaches allows us to evaluate differences and compare the previous version, JENA.V2, with JEAD. From the metrics, we will focus on [86]:

- Precision,
- Recall,
- F1 score,
- Macro average for each measure.

1. The matrix is composed of True Positives (TPs), False Positives (FPs), True Negatives (TNs), False Negatives (FNs).

In real-world scenarios, the clean code samples occur much more often than anti-pattern samples. As a result, our ground-truth dataset is highly imbalanced, with the majority of samples being clean and only a few being anti-pattern samples. The most informative metric is the Macro F1 score. The F1 score is the harmonic average of Precision and Recall. The macro F1 score is the average of the scores for both classes (in binary classification), mitigating the imbalance of the classes by giving equal weight to each class [86].

For the TECH_DEBT model, we provide a confusion matrix computed during training (evaluated on the test partition of the dataset), along with the metrics presented earlier. To illustrate the difference between training metrics and a sample in an open-source project, we count TPs and FPs and present the results in percentages. Due to the large number of comments in the projects, creating a ground-truth dataset would be a high effort with little benefit.

7.3 (DI8) God Dependency Injection Anti-pattern

The results for this model show that the success rate of detecting an anti-pattern varies from project to project. For Klaw [84], the model successfully detected one more anti-pattern sample than the rule-based detector, but produced more FPs. On the other hand, rule-based detectors detected more positive samples in the Apollo [85], again producing almost 2 times more FPs. The confusion matrix is presented in Table 7.1.

According to the performance measures presented in Table 7.2 and Table 7.3, both approaches are very comparable and yield similar results, as the F1 scores in each project are similar across the two detection strategies. The average precision and recall metrics show that the ML approach generalises and adapts across different code bases, as the difference between those metrics across projects is lower than for rule-based approaches. This means that anti-patterns detected in the Apollo project are more closely aligned with the defined static threshold than in Klaw, leading to more positive items being detected. This threshold can, of course, be adjusted through configuration, which is not possible for the ML model. On the contrary, the model uses more metrics to decide whether the provided class is an anti-pattern, which might

explain better generalisation across codebases. In conclusion, based on these results, it is not possible to say which approach is generally better, mainly because the number of anti-patterns in both projects was low, but ML models slightly outperform in terms of stability.

Table 7.1: Confusion matrix of DI8 anti-pattern detection per project

Project	Approach	TP	TN	FP	FN	Total
Klaw	ML	4	85	3	7	99
	Rule-based	3	87	0	8	99
Apollo	ML	6	150	7	3	166
	Rule-based	9	144	13	0	166

Table 7.2: DI8 anti-pattern detectors performance measures comparison (project Klaw)

Approach	Anti-pattern	Precision	Recall	**F1-score**	Items
ML	Clean code	0.92	0.97	0.94	92
	DI8 code	0.57	0.36	0.44	7
	macro average	0.75	0.66	0.69	99
Rule-based	Clean code	0.92	1	0.96	96
	DI8 code	1	0.27	0.42	3
	macro average	0.96	0.64	0.69	99

7.4 (SER1) Multi-service Anti-pattern

As presented in the confusion matrix (Table 7.4), the ML approach dramatically outperforms the rule-based detector in FPs. It appears that the static threshold based on the number of classes can detect most positive issues, but also produces many FPs. On the contrary, the ML approach, which is based on the semantic clustering of method

Table 7.3: DI8 anti-pattern detectors performance measures comparison (project Apollo)

Approach	Anti-pattern	Precision	Recall	**F1-score**	Items
ML	Clean code	0.99	0.96	0.97	153
	DI8 code	0.46	0.67	0.55	13
	macro average	0.72	0.81	0.76	166
Rule-based	Clean code	1	0.92	0.96	144
	DI8 code	0.41	1	0.58	22
	macro average	0.70	0.96	0.77	166

names (see Section 6.3), was not able to detect all cases and missed more cases than the rule-based approach.

The results are a bit different from project to project. According to the F1 score, the rule-based approach has a slight advantage over ML in Klaw project (see Table 7.5). It detected more and missed fewer classes, but produced more FPs. This implies that the project has fewer classes with more methods, which the static detector marked as issues. But the methods might be more semantically similar, or the difference in vector embeddings might not have been so obvious, so the clustering algorithm did not mark them as clustered. For the Apollo project (Table 7.6), the rule-based analyser produced a large number of FPs, while the ML detector produced significantly fewer. This means that the Apollo project has many classes with more methods marked as issues by the static threshold, but the methods are semantically similar. In conclusion, the ML models show better generalisation across projects but do not detect all possible issues. But it significantly outperforms the rule-based analyser in producing fewer FPs, especially on project Apollo. This increases the reliability of the produced issues. The rule-based approach is less adaptable to different code structures and would require adjustments to the static threshold.

Table 7.4: Confusion matrix of SER1 anti-pattern detection per project

Project	Approach	TP	TN	FP	FN	Total
Klaw	ML	3	54	0	6	63
	Rule-based	5	51	3	4	63
Apollo	ML	2	99	1	0	102
	Rule-based	2	82	18	0	102

Table 7.5: SER1 anti-pattern detectors performance measures comparison (project Klaw)

Approach	Anti-pattern	Precision	Recall	**F1-score**	Items
ML	Clean code	0.9	1	0.95	60
	SER1 code	1	0.33	0.50	3
	macro average	0.95	0.67	0.72	63
Rule-based	Clean code	0.93	0.94	0.94	57
	SER1 code	0.625	0.56	0.59	6
	macro average	0.78	0.75	0.76	63

7.5 (TECH_DEBT) Technical Debt Anti-patterns

Since the TECH_DEBT1 and TECH_DEBT2 anti-patterns are implemented using a single ML model, the evaluation is combined into a single chapter. Also, the comparison with the rule-based detector is omitted, because the anti-patterns were defined in this thesis, and no detector was developed in JENA.V2 [4].

The Table 7.7 shows the model's TPs and FPs statistics, along with the total number of found issues and the number of available comment samples in the project. The comment sample does not refer to an individual comment, but rather to the samples after extraction and processing by JEAD (see Section 6.4).

The TECH_DEBT model achieved high performance during training and evaluation on the test partition of the dataset with an F1 score

Table 7.6: SER1 anti-pattern detectors performance measures comparison (project Apollo)

Approach	Anti-pattern	Precision	Recall	**F1-score**	Items
ML	Clean code	1	0.99	0.99	99
	SER1 code	0.67	1	0.8	3
	macro average	0.83	0.99	0.90	102
Rule-based	Clean code	1	0.82	0.90	82
	SER1 code	0.1	1	0.18	20
	macro average	0.55	0.91	0.54	102

of 0.93 and a precision of approximately 92.9% (see Table 7.8). However, while evaluating on project Klaw, the precision for TECH_DEBT1 achieved only 22.4 % with a false positive rate of 77.6 %. The results showed a precision of 50%, slightly better than before, for the label TECH_DEBT2. The evaluation of the Apollo project shows an even greater drop in precision and an increase in the false-positive rate of the TECH_DEBT1 model, reaching almost 92%. These results show the model does not perform well in generalisation across different codebases. It performed very well on the training dataset, but because the set of open-source projects used to train the model was relatively small (around 3000 samples per label), the model was slightly overfit to the patterns in those projects.

The identification of Software Anti-patterns related to SATD varies significantly across different codebases. Each codebase employs unique markers to indicate bugs, including differences in naming conventions, comment structures, and terminology. One common issue is that the model often misclassifies comments containing phrases associated with the TECH_DEBT1 label, such as “it should…”, “create…”, and “delete…". These phrases typically convey that something is incomplete or requires future work. However, they can also appear in comments that are contextually “clean” and do not indicate any issues. As a result, the dataset used to train the model may be imbalanced in its representation of these terms, potentially leading to a high

false-positive rate. The future enhancement of this model should include the creation of a high-quality, balanced dataset. The corpus of open-source projects would need to be larger, with much greater effort devoted to labelling and dataset processing.

FPs produced with a commented code label (TECH_DEBT2) often included comments explaining some parts of the code, together with the code snippets. For example, snippets in the project Apollo included a textual visualisation of the release process feature described in the unit tests. Another FP included examples of properties used in the configuration files, which are more like explanatory, clean comments than code snippets. It is necessary to identify such cases in the dataset's clean labels so the model can correctly recognise patterns and reduce the false-positive rate.

In future work, it should also be considered to split the model into two separate models, one for each technical debt anti-pattern. The primary motivation for multi-label classification was that a code snippet might include both commented-out code and technical debt, but these examples are rare (at least in the projects we used for training). This way, the model could learn to correct the opposite clean patterns for each anti-pattern more precisely. In contrast, this adds another layer of complexity to training and data preprocessing and increases the need for more resources for model deployment.

Table 7.7: True and False positives statistics of TECH_DEBT model

Project	Anti-p.	TP	FP	TP + FP	TP (%)	FP (%)	Total
Klaw	TD1	13	45	58	22.4	77.6	2009
	TD2	16	16	32	50.0	50.0	
Apollo	TD1	15	170	185	8.1	91.9	2418
	TD2	9	59	86	13.2	86.8	

7.6 Validity and Applicability of the Models

The results indicate that machine learning models are suitable for detecting structural and semantic anti-patterns in Java enterprise systems, offering improved adaptability across different codebases. How-

Table 7.8: TECH_DEBT model performance on test dataset during training

Epoch	Precision	Recall	F1
1	0.9138	0.9054	0.9095
2	0.9142	0.9156	0.9147
3	0.9290	0.9344	0.9316

ever, rule-based approaches remain competitive, particularly in terms of recall and interpretability. The findings suggest that neither approach is universally superior, and that combining them provides the most robust solution. A hybrid strategy, where machine learning models are used to improve generalisation and reduce false positives while rule-based detectors ensure coverage, appears to be the most appropriate approach for practical anti-pattern detection in enterprise environments.

Key Findings

God DI

- Machine learning and rule-based approaches achieve comparable performance, with similar F1 scores across both projects, making it difficult to declare a clear winner.
- The ML model shows better stability and generalisation across different codebases by using multiple features rather than fixed thresholds.
- Rule-based detection can sometimes achieve higher recall but is sensitive to threshold configuration and produces more inconsistent results between projects.

Multi-service

- The ML model significantly reduces false positives, making its detections more reliable, especially in projects like Apollo.
- However, it has lower recall, missing some true anti-pattern cases compared to the rule-based approach.
- Rule-based detection finds more issues but produces many false positives and requires tuning, while ML offers better generalisation across different project structures.

Technical Debt

- The model achieves high performance during training (F1 $\approx$ 0.93), but this does not transfer to real-world projects.
- In practice, it shows very high false-positive rates and poor precision, indicating weak generalisation.
- The main issues are overfitting and imbalanced training data, especially due to variability in natural-language comments across codebases.

7.7 Threats to Validity

One validity threat is the manual labelling of both the training and ground-truth datasets for evaluation. Evaluating code snippets and metrics is often subjective because it is hard to determine the correct boundary between when a code snippet is an anti-pattern and when it is not. This may lead to inconsistencies across the evaluated projects, with some developers seeing the evaluation differently.

Another threat is the limited number of projects used for model training, especially for the TECH_DEBT model, resulting in a small, tightly coupled dataset. The chosen approach involves significant manual labelling of the dataset, which is limited in the thesis due to other work on model deployment and detection infrastructure. The labelling is often performed by a group of experienced developers, or using the Crowd-smelling [35] approach (see Section 3.2.2). This approach enables better diversification of the dataset across various projects, allowing the model to generalise the identified snippets more effectively.

One key factor is reproducibility. The electronic attachment of the thesis includes all necessary scripts, source code, and datasets for training the models, conducting manual evaluation, or running the evaluation. However, ML models (especially the TECH_DEBT model) require a GPU for training; otherwise, training may take several hours or be impossible on a CPU. To mitigate this threat, the finetuned models are included in the thesis attachment and are ready for deployment on the Inference Server. Using a GPU for inference is optional but recommended, as it significantly improves performance.

8 Conclusion

This thesis aimed to refactor and extend the JENA static analysis detector developed in previous theses [3][4], enabling the use of ML models for the detection of Enterprise Java anti-patterns. The new version of the tool is called JEAD.

We implemented an infrastructure that supports sending code features for evaluation by ML models. This infrastructure includes multiple changes and improvements on the JEAD side, for example, the implementation of code extractors, inference queues, and extended configuration, which allow dynamic registration of models. We also improved JEAD by adding support for the Lombok code generation library and introducing custom JEAD Maven and Gradle plugins. A significant improvement to the project's structure made future development clearer for developers. The user experience was also enhanced by the zip package, automatically generated by GitHub Actions and eliminating the need to rebuild the project each time.

The infrastructure also includes implementation of the Inference Server, which can dynamically load ML models and expose them through the unified gRPC interface. JEAD can connect to this server and send code features for evaluation. ML models are included and loaded as plugins, enabling customisation and easy addition of models. The server is released as a Docker image and can be run with Docker Compose scripts provided in JEAD's zip package. The source code is published in a GitHub repository[1]. Created Docker Images are published on DockerHub[2].

We also defined and prepared three ML solutions that demonstrate their potential to detect Java Enterprise Anti-patterns. The results show that they are comparable to rule-based detectors and, in some cases, even better. The DI8 and SER1 models showed better stability and generalisation across the evaluated codebases than the corresponding rule-based detectors. However, the TECH_DEBT model did not generalise well to the evaluated projects and produced many false positives. This is mainly due to the limitations of the thesis and to a smaller

1. https://github.com/Ricaps/JEAD
2. https://hub.docker.com/r/ricaps/jead

number of projects selected for the dataset, so the model does not generalise as well across other code bases.

In conclusion, ML models show great potential for detecting enterprise anti-patterns. However, there are cases where rule-based detection might be sufficient, mainly due to the effort required to train models, infrastructure needs, and the performance and resource requirements for model deployment. Models often require more data and effort to achieve better performance and reliability. Also, the models are often "black boxes", so they might not be trusted by stakeholders and developers, leading to the analysis results being ignored.

Future work should aim to define more ML solutions using prepared infrastructure. Our thesis selected a few anti-patterns from the catalogue to demonstrate the possibilities of machine learning (or artificial intelligence in general). There should also be a focus on improving the Inference Server in terms of security, automatic models undeployment, and API improvements.

Bibliography

1. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN 9780321700698.
2. BROWN, William H.; MALVEAU, Raphael C.; MCCORMICK, Hays W.; MOWBRAY, Thomas J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. John Wiley & Sons Ltd, 1998. ISBN 978-0-471-19713-3.
3. BATEĽ, Jakub. *Catalogue of Anti-Patterns in Java Enterprise Development*. Brno, 2022. Master's Thesis. Masaryk University, Faculty of Informatics.
4. MIČKA, Petr. *Anti-patterns Detection in Java Software Enterprise Development*. Brno, 2024. Master's Thesis. Masaryk University, Faculty of Informatics.
5. *Project Lombok* [online]. 2025. [visited on 2026-02-10]. Available from: `https://projectlombok.org/`.
6. GAFFNEY JR, John E. Metrics in software quality assurance. In: *Proceedings of the ACM'81 conference*. 1981, pp. 126–130.
7. GREEN, Collin; JASPAN, Ciera; HODGES, Maggie; LIN, Jessica. Developer Productivity for Humans, Part 7: Software Quality. *IEEE Software*. 2024, vol. 41, no. 1, pp. 25–30. Available from DOI: `10.1109/MS.2023.3324830`.
8. MCCALL, Jim A; RICHARDS, Paul K; WALTERS, Gene F. *Factors in software quality. volume i. concepts and definitions of software quality*. 1977. Tech. rep.
9. PALOMBA, Fabio; ZANONI, Marco; FONTANA, Francesca Arcelli; DE LUCIA, Andrea; OLIVETO, Rocco. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*. 2017, vol. 45, no. 2, pp. 194–218.

10. CHIDAMBER, S.R.; KEMERER, C.F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*. 1994, vol. 20, no. 6, pp. 476–493. Available from DOI: 10.1109/32.295895.

11. BELLIN, David; TYAGI, Manish; TYLER, Maurice. Object-oriented metrics: an overview. In: *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*. Toronto, Ontario, Canada: IBM Press, 1994, p. 4. CASCON '94.

12. *Software engineering — Product quality — Part 1: Quality model*. Geneva, Switzerland: International Organization for Standardization and International Electrotechnical Commission, 2001. No. ISO/IEC 9126-1:2001. Available also from: https://www.iso.org/standard/22749.html. Withdrawn standard.

13. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model overview and usage*. Geneva, Switzerland: International Organization for Standardization and International Electrotechnical Commission, 2024. No. ISO/IEC 25002:2024. Available also from: https://www.iso.org/standard/78175.html.

14. BUSCHMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAD, Peter; STAL, Michael. *Pattern-oritented Software Architecture*. A System of Patterns. John Wiley & Sons Ltd, 2001. ISBN 9780471958697.

15. LANZA, Michele; MARINESCU, Radu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.

16. FOOTE, Brian; YODER, Joseph. Big ball of mud. *Pattern languages of program design*. 1997, vol. 4, pp. 654–692.

17. RIVERA, Richard; MOSQUERA, Marcela; FLORES, Pamela; ANCHUNDIA, Carlos E.; JIMÉNEZ, Alejandro; CARPIO, Xavier. Smells vs. Software Antipatterns: Do Definitions Matter? In: *2025 IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA)*. 2025, pp. 169–174. Available from DOI: 10.1109/SERA65747.2025.11154518.

18. POLU, Omkar Reddy. AI-Driven Automatic Code Refactoring for Performance Optimization. *International Journal of Science and Research*. 2025, vol. 14, no. 1, pp. 1316–1320.

19. CORDEIRO, Jonathan; NOEI, Shayan; ZOU, Ying. LLM-Driven Code Refactoring: Opportunities and Limitations. In: *2025 IEEE/ACM Second IDE Workshop (IDE)*. IEEE, 2025, pp. 32–36.

20. GEZICI GEÇER, Bahar; KOLUKISA TARHAN, Ayça. Explainable AI framework for software defect prediction. *Journal of Software: Evolution and Process*. 2025, vol. 37, no. 4, e70018.

21. MOHAPATRA, Partha Sarathi. *Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle*. Deep Science Publishing, 2025. Available from DOI: 10.70593/978-93-7185-046-9.

22. FERREIRA, Kecia A.M.; BIGONHA, Mariza A.S.; BIGONHA, Roberto S.; MENDES, Luiz F.O.; ALMEIDA, Heitor C. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*. 2012, vol. 85, no. 2, pp. 244–257. ISSN 0164-1212. Available from DOI: 10.1016/j.jss.2011.05.044. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

23. *PMD - source code analyzer* [online]. PMD Open Source Project, 2026. [visited on 2026-02-19]. Available from: https://docs.pmd-code.org/latest/index.html.

24. *ASM framework* [online]. INRIA, France Telecom, 2025. [visited on 2026-02-19]. Available from: https://asm.ow2.io/index.html.

25. *Spring Shell* [online]. Broadcom, 2026. [visited on 2026-02-20]. Available from: https://docs.spring.io/spring-shell/reference/index.html.

26. *Spring Boot* [online]. Broadcom, 2026. [visited on 2026-02-20]. Available from: https://docs.spring.io/spring-boot/index.html.

27. *Java Parser* [online]. Github repository. 2026. [visited on 2026-02-20]. Available from: https://github.com/javaparser/javaparser.

28. *SonarQube Documentation* [online]. SonarSource Sàrl, 2025. [visited on 2026-02-19]. Available from: `https://docs.sonarsource.com/`.

29. *Code Quality and Security for Java* [online]. Github repository. SonarSource Sàrl, 2026. [visited on 2026-02-19]. Available from: `https://github.com/SonarSource/sonar-java`.

30. LAIGNER, Rodrigo Nunes. *Cataloging Dependency Injection Anti-Patterns in Software Systems*. 2020. Available from DOI: `10.17771/PUCRio.acad.47298`. MA thesis. Pontifical Catholic University of Rio de Janeiro (PUC-Rio).

31. FOKAEFS, Marios; TSANTALIS, Nikolaos; STROULIA, Eleni; CHATZIGEORGIOU, Alexander. JDeodorant: identification and application of extract class refactorings. In: *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 1037–1039. ICSE '11. ISBN 9781450304450. Available from DOI: `10.1145/1985793.1985989`.

32. ALI, Israr; RIZVI, Syed Sajjad Hussain; ADIL, Syed Hasan. Enhancing software quality with AI: A transformer-based approach for code smell detection. *Applied Sciences*. 2025, vol. 15, no. 8, p. 4559.

33. FENG, Zhangyin; GUO, Daya; TANG, Duyu; DUAN, Nan; FENG, Xiaocheng; GONG, Ming; SHOU, Linjun; QIN, Bing; LIU, Ting; JIANG, Daxin, et al. Codebert: A pre-trained model for programming and natural languages. In: *Findings of the association for computational linguistics: EMNLP 2020*. 2020, pp. 1536–1547.

34. GUO, Daya; REN, Shuo; LU, Shuai; FENG, Zhangyin; TANG, Duyu; LIU, Shujie; ZHOU, Long; DUAN, Nan; SVYATKOVSKIY, Alexey; FU, Shengyu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*. 2020.

35. REIS, José Pereira dos; ABREU, Fernando Brito e; CARNEIRO, Glauco de Figueiredo. Crowdsmelling: A preliminary study on using collective knowledge in code smells detection. *Empirical Software Engineering*. 2022, vol. 27, no. 3, p. 69.

36. HO, Anh; BUI, Anh MT; NGUYEN, Phuong T; DI SALLE, Amleto; LE, Bach. EnseSmells: Deep ensemble and programming language models for automated code smells detection. *Journal of Systems and Software*. 2025, vol. 224, p. 112375.

37. HO, Anh; BUI, Anh M. T.; NGUYEN, Phuong T.; DI SALLE, Amleto. Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. Oulu, Finland: Association for Computing Machinery, 2023, pp. 229–234. EASE '23. Available from DOI: 10.1145/3593434.3593476.

38. ANICHE, Maurício. *CK: Code metrics for Java code by means of static analysis* [online]. GitHub, 2022 [visited on 2026-02-22]. Available from: https://github.com/mauricioaniche/ck.

39. AZADI, Umberto; ARCELLI FONTANA, Francesca; ZANONI, Marco. *Machine Learning based Code Smell Detection through WekaNose*. 2018. Available from DOI: 10.1145/3183440.3194974.

40. CRUZ, Daniel; SANTANA, Amanda; FIGUEIREDO, Eduardo. Evaluating a continuous feedback strategy to enhance machine learning code smell detection. *Science of Computer Programming*. 2026, vol. 247, p. 103346. ISSN 0167-6423. Available from DOI: 10.1016/j.scico.2025.103346.

41. *Spring Framework Documentation* [online]. Broadcom, 2026. [visited on 2026-02-26]. Available from: https://docs.spring.io/spring-framework/reference/index.html.

42. RON PRESSLER, Alan Bateman. *JEP 444: Virtual Threads* [online]. OpenJDK, 2025. [visited on 2026-02-26]. Available from: https://openjdk.org/jeps/444.

43. ANICHE, Maurício. *gRPC: A high performance, open source universal RPC framework* [online]. 2026. [visited on 2026-02-26]. Available from: https://grpc.io/.

44. *Spring gRPC: GRPC Client* [online]. VMWare, 2025. [visited on 2026-02-26]. Available from: https://docs.spring.io/spring-grpc/reference/client.html.

45. SCOPES, Ashley. *Protobuf Maven Plugin* [online]. 2026. [visited on 2026-02-26]. Available from: https://ascopes.github.io/protobuf-maven-plugin/.

46. *Delombok* [online]. 2025. [visited on 2026-02-10]. Available from: https://projectlombok.org/features/delombok.

47. *Apache Maven Resources Plugin* [online]. 2026. [visited on 2026-03-10]. Available from: https://maven.apache.org/plugins/maven-resources-plugin/.

48. *GitHub Packages documentation* [online]. 2026. [visited on 2026-03-10]. Available from: https://docs.github.com/en/packages.

49. *Maven Assembly Plugin* [online]. 2026. [visited on 2026-03-11]. Available from: https://maven.apache.org/plugins/maven-assembly-plugin/.

50. *GitHub Actions* [online]. 2026. [visited on 2026-03-11]. Available from: https://github.com/features/actions.

51. *Docker* [online]. 2026. [visited on 2026-03-11]. Available from: https://www.docker.com/.

52. *Docker Compose* [online]. 2026. [visited on 2026-03-11]. Available from: https://docs.docker.com/compose/.

53. *NVIDIA Triton Inference Server* [online]. 2026. [visited on 2026-03-11]. Available from: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

54. *MLServer* [online]. 2026. [visited on 2026-03-11]. Available from: https://docs.seldon.ai/mlserver.

55. *KServe* [online]. 2026. [visited on 2026-03-11]. Available from: https://kserve.github.io/website/.

56. FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern* [online]. [visited on 2026-03-12]. Available from: https://martinfowler.com/articles/injection.html.

57. *asyncio – Asynchronous I/O* [online]. 2026. [visited on 2026-03-12]. Available from: https://docs.python.org/3/library/asyncio.html.

58. *Poetry: Python packaging and dependency management made easy* [online]. 2026. [visited on 2026-03-12]. Available from: https://python-poetry.org/.

59. *Python gRPC* [online]. 2026. [visited on 2026-02-26]. Available from: https://grpc.io/docs/languages/python/quickstart/.

60. *Requirements File Format* [online]. 2026. [visited on 2026-03-13]. Available from: https://pip.pypa.io/en/stable/reference/requirements-file-format/.

61. *pip: The PyPA recommended tool for installing Python packages.* [online]. 2026. [visited on 2026-03-13]. Available from: https://pypi.org/project/pip/.

62. *Docker Hub* [online]. 2026. [visited on 2026-03-13]. Available from: https://hub.docker.com/repository/docker/ricaps/jead/general.

63. *NVIDIA* [online]. 2026. [visited on 2026-03-13]. Available from: https://www.nvidia.com/.

64. *ONNX* [online]. 2026. [visited on 2026-03-13]. Available from: https://onnx.ai/.

65. *scikit-learn* [online]. 2026. [visited on 2026-03-13]. Available from: https://scikit-learn.org/stable/index.html.

66. *Transformers* [online]. Hugging Face, 2026. [visited on 2026-03-15]. Available from: https://huggingface.co/docs/transformers/index.

67. FONTANA, Francesca Arcelli; ZANONI, Marco; MARINO, Alessandro; MÄNTYLÄ, Mika V. Code Smell Detection: Towards a Machine Learning-Based Approach. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 396–399. Available from DOI: 10.1109/ICSM.2013.56.

68. ARCELLI FONTANA, Francesca; MÄNTYLÄ, Mika V; ZANONI, Marco; MARINO, Alessandro. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*. 2016, vol. 21, no. 3, pp. 1143–1191.

69. AZEEM, Muhammad Ilyas; PALOMBA, Fabio; SHI, Lin; WANG, Qing. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*. 2019, vol. 108, pp. 115–138.

70. *Code and Metrics* [online]. Hugging Face, 2026. [visited on 2026-03-15]. Available from: `https://github.com/yegor256/cam`.

71. *Pandas: Python Data Analysis Library* [online]. 2026. [visited on 2026-03-15]. Available from: `https://pandas.pydata.org/`.

72. *skl2onnx: Convert scikit-learn models to ONNX* [online]. 2025. [visited on 2026-03-21]. Available from: `https://pypi.org/project/skl2onnx/`.

73. SIVAKUMAR, Amrutha; BHAVYA, G; M, Srinidhi; NARMADA, Nakka; PATI, Peeta Basa. Clustering and Visualizing Programming Solutions: A Transformer-Based Analysis Using CodeBERT and CodeT5+. In: *2025 International Conference on Emerging Technologies in Computing and Communication (ETCC)*. 2025, pp. 1–6. Available from DOI: 10.1109/ETCC65847.2025.11108390.

74. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction* [online]. 2026. [visited on 2026-03-21]. Available from: `https://umap-learn.readthedocs.io/en/latest/`.

75. *scikit-learn Clustering: HDBSCAN* [online]. 2026. [visited on 2026-03-21]. Available from: `https://scikit-learn.org/stable/modules/clustering.html#hdbscan`.

76. DENG, Dingsheng. DBSCAN Clustering Algorithm Based on Density. In: *2020 7th International Forum on Electrical Engineering and Automation (IFEEA)*. 2020, pp. 949–953. Available from DOI: 10.1109/IFEEA51475.2020.00199.

77. LI, Yikun; SOLIMAN, Mohamed; AVGERIOU, Paris. Automatically Identifying Relations Between Self-Admitted Technical Debt Across Different Sources. In: *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. 2023, pp. 11–21. Available from DOI: 10.1109/TechDebt59074.2023.00008.

78. MAHMOOD, Sajjad. Detecting Inline Code Comment Smells Leveraging CodeBERT Model. *IEEE Access*. 2026, vol. 14, pp. 28367–28382. Available from DOI: 10.1109/ACCESS.2026.3666288.

79. *Apache Hive (TM)* [online]. 2026. [visited on 2026-03-21]. Available from: https://github.com/apache/hive.

80. *Apache Tomcat* [online]. 2026. [visited on 2026-03-21]. Available from: https://github.com/apache/tomcat.

81. *ThingsBoard* [online]. 2026. [visited on 2026-03-21]. Available from: https://github.com/thingsboard/thingsboard.

82. *PA165 Java Enterprise Applications* [online]. Faculty of Informatics, Masaryk University, 2024. [visited on 2026-03-21]. Available from: https://is.muni.cz/predmet/fi/jaro2024/PA165.

83. *onnxruntime* [online]. Microsoft, 2026. [visited on 2026-03-21]. Available from: https://pypi.org/project/onnxruntime/.

84. *Klaw* [online]. Aiven, 2026. [visited on 2026-03-06]. Available from: https://github.com/Aiven-Open/klaw.

85. *Apollo: A reliable configuration management system* [online]. 2026. [visited on 2026-03-06]. Available from: https://github.com/apolloconfig/apollo.

86. AMIN, Fahmy; MAHMOUD, M. Confusion matrix in binary classification problems: A step-by-step tutorial. *Journal of Engineering Research*. 2022, vol. 6, no. 5.

87. FOWLER, Martin; BECK, Kent. *Refactoring: improving the design of existing code*. Second Edition. Boston: Addison-Wesley, 2019. ISBN 978-0-13475759-9.

88. POTDAR, Aniket; SHIHAB, Emad. An Exploratory Study on Self-Admitted Technical Debt. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 91–100. Available from DOI: 10.1109/ICSME.2014.31.

A Java Enterprise Anti-pattern Catalogue

The following sections present a catalogue of Java Enterprise Anti-patterns based on the Master's thesis by Mička [4]. This thesis expanded the catalogue by adding the Technical Debt section. Additionally, we analyzed the suitability of all anti-patterns for machine learning detection. For anti-patterns that lack analysis text, they are not suitable for detection by machine learning models.

A.1 Dependency Injection

Name	*DI1 Unused Injection*
Description	A dependency is injected into the bean, but it is unused. It is very similar to the unused field anti-pattern.
ML Detection	–
Name	*DI2 Direct Container Call*
Description	A bean calls the methods of the DI container to inject a dependency. It should use an abstraction, not a direct call.
ML Detection	–
Name	*DI3 Concrete Class Injection*
Description	The injected type is a concrete class, instead of an interface or an abstract class.
ML Detection	–
Name	*DI4 Open Window Injection*
Description	Other classes can access the injected dependency. It is either passed to some other classes or stored in the public field.
ML Detection	–

Name	*DI5 Framework Coupling*
Description	Using framework-specific annotations such as Autowired or Service instead of generic ones from the Jakarta specifications, like Inject. It can make it harder to swap between Dependency Injection frameworks.
ML Detection	–

Name	*DI6 Multiple Forms Of Injection*
Description	The same dependency is injected using multiple injection methods. For example, using setters and fields. This can cause multiple side effects, and the code might become unreadable.
ML Detection	–

Name	*DI7 Open Door Injection*
Description	A injected dependency stored in a non-final field can be overridden by the new field assignment, for example, by the public setter.
ML Detection	–

Name	*DI8 God DI Class*
Description	The bean has injected too many fields and does a lot of unrelated stuff. It is a variant of the Large class anti-pattern defined by Fowler [87].
ML Detection	Traditional ML techniques like Random Forests or Naive Bayes can help dynamically learn thresholds based on code metrics.

Name	*DI9 Multiple Assigned Injections*
Description	The injected dependency is stored in multiple fields. For example, the dependency can be stored in the current class's field and passed to the parent class via the constructor.
ML Detection	–

Name	*DI10 Long Producer Method*
Description	A method creating a bean (typically with a Bean annotation) is long and complex. It also has more responsibilities than producing a bean, such as enforcing business rules.
ML Detection	ML techniques could learn dynamic thresholds for the detection of method complexity. Some NLP models could understand the code and detect complex logic.

Name	*DI11 Intransigent Injection*
Description	A performance-heavy dependency is injected eagerly at bean construction, instead of lazily loading when needed.
ML Detection	ML model could learn patterns typical for performance-heavy calls and help identify such beans.

Name	*DI12 Static Dependence Provider*
Description	The class is getting dependencies using static methods. Such dependency might use the singleton design pattern, for example. It results in tight coupling between classes.
ML Detection	–

A.2 Persistence Layer

Name	*PER1 N+1 Query Problem*
Description	The anti-pattern may occur when we have associated entities (many-to-one, one-to-many) with eager fetching of related data. In this case, N queries are used instead of a single query to fetch data.
ML Detection	–

Name	*PER2 Dredge*
Description	This anti-pattern occurs when we use deep queries to fetch entities from the database. It might end up loading the entire graph of entities, causing real performance problems.
ML Detection	Perhaps some graph-aware model with entity context could detect heavy database queries.

Name	*PER3 Crush*
Description	Data integrity is lost when a code operates on invalid data it loaded before another unrelated code changed it. Typically, this happens when entity A loads related data X; entity B also loads data X and modifies it; then entity A works with outdated data.
ML Detection	–

A.3 Service Layer

Name	*SER1 Multi-service*
Description	Service consists of unrelated methods and can be split into multiple interfaces. It uses a method for different use cases and different types.
ML Detection	More clusters of semantically similar method names could reveal multi-service anti-patterns.

Name	*SER2 Tiny Service*
Description	There are multiple related services, each one with a small number of methods. It makes sense to merge them into services with meaningful sizes.
ML Detection	Semantic similarity of methods across multiple interfaces could help identify mergeable interfaces.

Name	*SER3 The Client Completes a Service*
Description	A service is missing important functionality, such as validation or security checks. Clients using such a service have to implement the missing functionality themselves instead of using the service.
ML Detection	NLP model could be used to learn patterns of missing functionality (validation, ...) and help identify places where it is missing.

A.4 Mocking

Name	*MOC1 Final Method Call with Exception*
Description	In the tested method, there is a call to a final method or a call to a method from a final class, which does functionality we would like to mock. It is harder and slower to mock the final methods.
ML Detection	–

Name	*MOC2 Constructor Call with Exception*
Description	There is a constructor call in a method we would like to test, and this constructor call can throw an exception and communicate with an external service we want to mock. The constructor call prevents mocking the layer communication with external services.
ML Detection	–

Name	*MOC3 Static Call with Exception*
Description	The tested method calls a static method we would like to mock because it communicates with an external service. The code using static methods is tightly coupled and less testable.
ML Detection	–

Name	*MOC4 Inappropriate Method Mocking*
Description	Mock maker editing byte code is used, for example, Mock maker, inline or Powermock, to mock static or final methods instead of refactoring.
ML Detection	–

A.5 Security

Name	*SEC1 Storing Secrets in Insecure Places*
Description	Storing sensitive information like passwords or usernames in insecure locations, such as configuration files or resources committed to version control systems.
ML Detection	Secrets can be saved in many forms. An NLP model could identify secrets leaked to the version control.

Name	*SEC2 Disabling CSRF Protection*
Description	The CSRF protection is turned off, for example, in Spring Security. It allows attackers to send malicious requests from their website to the honest website. It is needed especially for session-based authentication.
ML Detection	–

Name	*SEC3 Lifelong Access Tokens*
Description	The lifetime of access tokens is too long – hours, days. Access tokens should be short-lived and used together with refresh tokens.
ML Detection	–

Name	*SEC4 Insecure Default Configuration*
Description	A default configuration that is used is insecure. Some password encoders might use low security by default, so they need to be reconfigured.
ML Detection	–

Name	*SEC5 Signing JWT With Fixed Secret*
Description	JWT are signed with fixed secrets instead of random secrets with high entropy.
ML Detection	–

Name	*SEC6 Insecure Communication*
Description	TLS is not used for HTTP communication, so the communication is not encrypted.
ML Detection	–

A.6 Technical Debt

Name	*TECH_DEBT1 Self Admitted Technical Debt*
Description	The self-admitted technical debt often occurs as code comments, with patterns as *TODO*, *FIXME*, *to be done*, *refactor it*. It indicates that something in the code does not work properly, is a workaround, is incomplete, and needs future attention. Having such an anti-pattern in the enterprise codebase is not safe, since it means the software is most probably not working reliably [88].
ML Detection	Some patterns are obvious, and e.g. regular expressions are sufficient. But some NLP models could semantically understand a comment and detect SATD.

Name	*TECH_DEBT2 Commented Out Code*
Description	A healthy codebase should not include commented-out code. It often leads to maintainability and readability issues. The later use of such code might lead to the usage of untested, unsafe code. Also, the commented-out code often means missing functionality or a commented-out automated test. It is a better practice to remove dead code when using a proper version control system, so the history of removed code can be recovered later.
ML Detection	Code patterns can be partially detected using regular expressions. ML models can improve the identification of comments, which include code samples for informative purposes, not commented-out code.

B JEAD Project Structure

The following file tree presents the high-level structure of the JEAD project and its sub-projects. For simplicity, it omits concrete files.

```
JEAD/ ..........................................Root directory
├── .github/ ....................GitHub Workflow definitions
├── ai-detector/ .....Root directory for Python ML projects
│   ├── inference-server/ .......Inference Server sub-project
│   └── training/ ................Code for training ML models
├── assembly/ ......Maven project for maven-assembly-plugin
├── build-plugins/ ....................Custom build plugins
│   ├── jead-gradle-plugin/ ..............JEAD Gradle Plugin
│   ├── jead-maven-plugin/ ................JEAD Maven Plugin
│   └── jead-plugin-core/ ......Shared logic for both plugins
├── detector/ ..........................Static detector tool
└── test-fixtures/ .............Test fixtures for auto-tests
    ├── antipatterns/
    ├── authorization-server/
    ├── power-mock-usage/
    └── test-gradle-project/
```

C Running the Evaluation

All necessary steps are described in the JEAD runtime assembly package in `README.md` file. The following text describes the basic steps for configuring and running the application and the evaluation. The listed commands may differ across operating systems.

To run the evaluation using JEAD with the support of machine learning, you must meet the following prerequisites:

1. Obtain the JEAD runtime assembly package (see Appendix E).
2. Obtain the models root package (or create your own models; see Appendix E).
3. Install Java with version $>= 25$.
4. Install Docker with Docker Compose (or an alternative like Podman).
5. Configure .env file in the JEAD runtime assembly (path to models root and GPU availability flag)

Then, you have to perform the necessary configuration:

1. Unzip the JEAD runtime assembly package.
2. Unzip the models root folder and copy its absolute path.
3. Open the `.env` file in the text editor and set:
   - `MODELS_ROOT_HOST` to the absolute path to the models root folder.
   - `USE_GPU` to true in case you want to install GPU dependencies.
4. Using your GitHub account, generate a classic Personal Access Token (PAT)[1], with minimum permission `read:packages`.

1. `https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens#creating-a-personal-access-token-classic`

5. Open the `config/application.yml` file and insert the PAT to `prepare-plugin.repository.access-token` property.

To run the Inference Server, PostgreSQL database and JEAD detector, you have to use the following commands:

Listing C.1: Commands to run the application

```
# To run in the GPU mode, use:
docker compose -f compose.yaml -f compose.gpu.yaml up

# Or to run in CPU-only mode, use:
docker compose -f compose.yaml up

# To start JEAD detector application, use:
java -jar detector.jar
```

To run the evaluation on the project which is using Maven, you should perform the following steps (let's assume the evaluated project is located at `/home/projects/spring-project`:

1. Enter prepare command to the JEAD CLI:
   `prepareProjects -d /home/projects/spring-project`
2. Build the Maven project (run in the project's directory):
   `cd /home/projects/spring-project & mvn clean package`
3. Run the evaluation using JEAD CLI:
   `detectIssues -p /home/projects/spring-project`

D Architecture Diagrams

This appendix includes diagrams related to the JEAD and Inference server architectures.

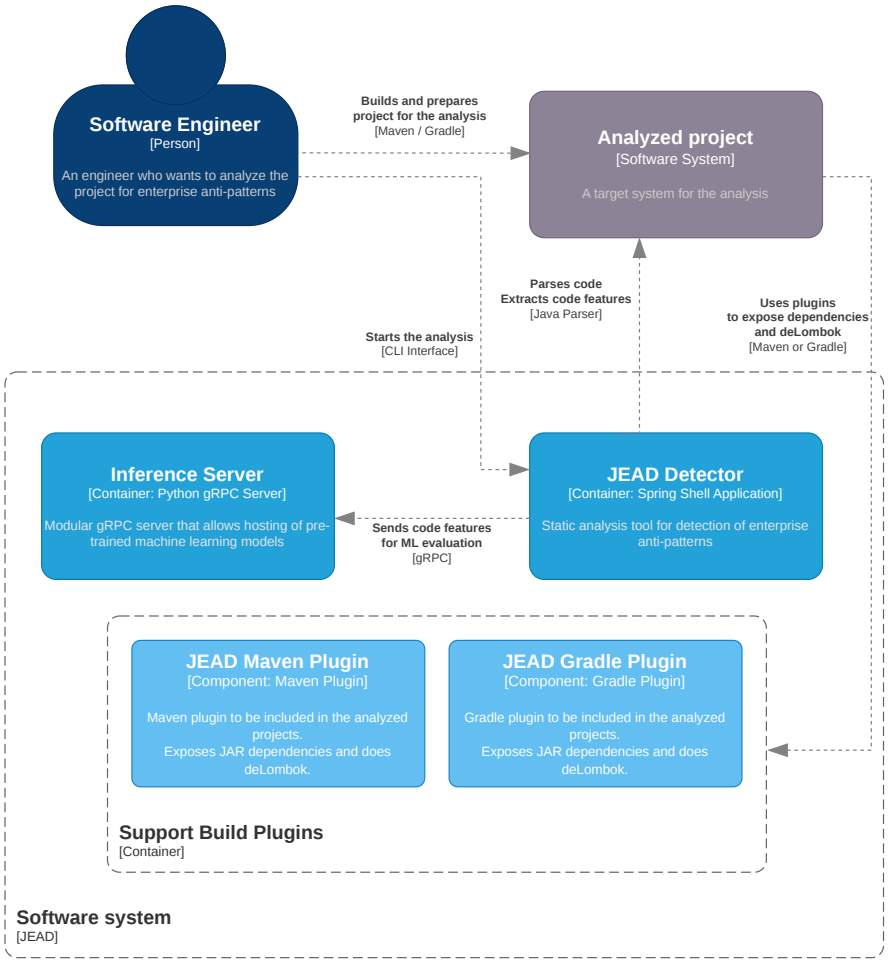


Figure D.1: System-level diagram of JEAD architecture

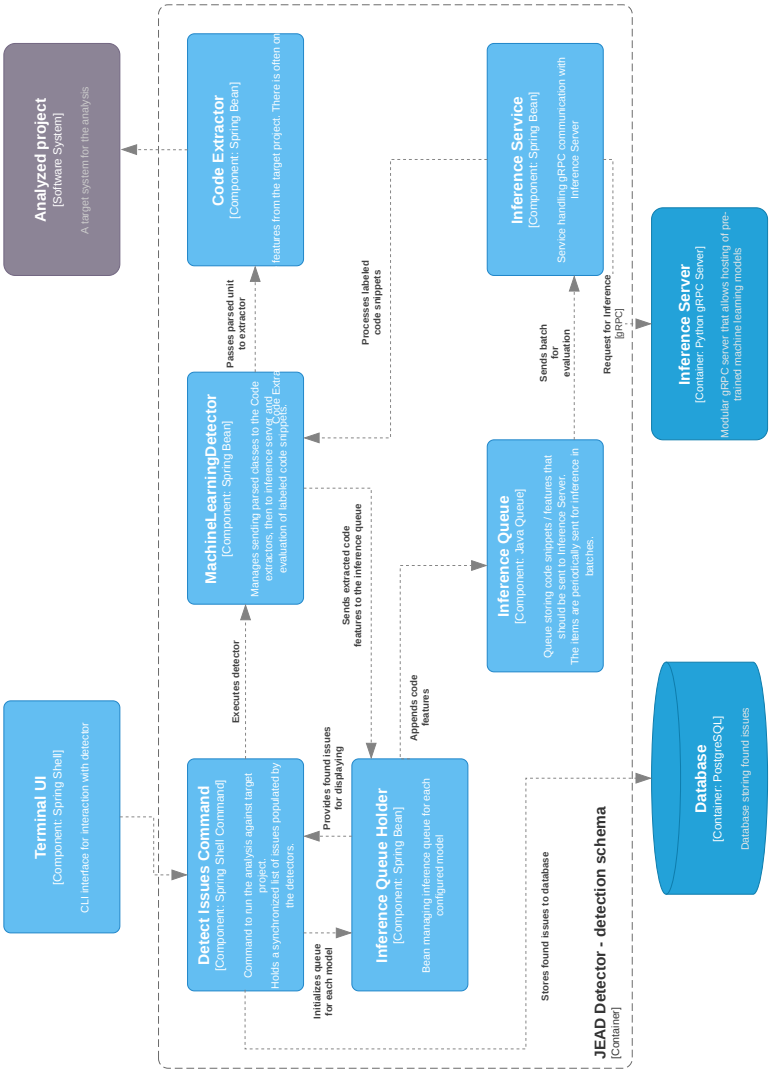


Figure D.2: Component-level diagram of JEAD architecture

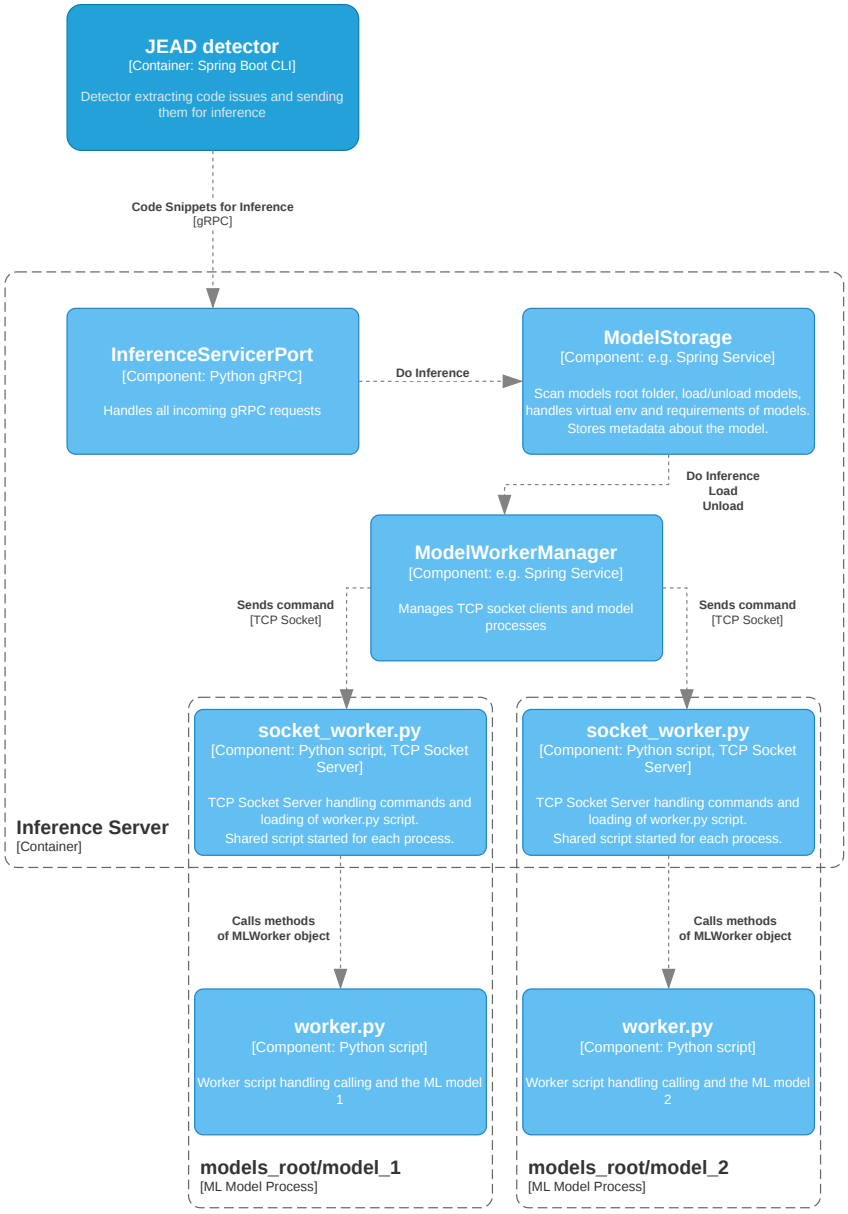


Figure D.3: Component-level diagram of Inference Server modular architecture

E Electronic Attachment

The following structure represents the folders and files included in the thesis electronic attachment.

```
submission-thesis/
├── README.md ........ Documentation of the bundle structure
├── Jead-master/ ..................... Source code repository
├── datasets/ ..................... Arrow and JSONL datasets
├── evaluation/ ............. Models evaluation in ODS format
├── jead-runtime/ .................. Runtime assembly package
└── models_root/ ... Models root including trained ML models
```