University of Minho
School of Engineering

André Filipe Araújo Freitas

Enhancing transpilers with machine learning techniques

october 2025

University of Minho
School of Engineering

André Filipe Araújo Freitas

Enhancing transpilers with machine learning techniques

Master's Dissertation in Informatics Engineering

Dissertation supervised by
Pedro Rangel Henriques
Tiago Baptista

october 2025

Acknowledgements

I wish to express my sincere gratitude to Professor Pedro Rangel Henriques and Tiago Baptista, coordinator and supervisor respectively, for their guidance, availability, and continuous support throughout the development of this work. Their expertise, constructive feedback, and commitment were instrumental to the successful completion of the project.

Statement of Integrity

University of Minho, Braga, october 2025

André Filipe Araújo Freitas

Abstract

In recent years, the rapid rise of Large Language Models (LLMs) has transformed multiple domains, inspiring new possibilities for automating complex cognitive tasks — including software engineering. This work explores the potential of leveraging LLMs to accelerate the migration of legacy codebases to modern environments. Traditional transpilers, which convert source code from one language to another, have long relied on rule-based approaches and Abstract Syntax Tree (AST) transformations. While these methods are reliable for syntactic conversions, they often struggle with contex -sensitive constructs, framework-specific logic, and maintaining semantic fidelity between distinct programming paradigms. To address these challenges, this research integrates machine learning techniques into a contemporary transpiler architecture, creating a novel hybrid system that combines the structural precision of AST-based transpilation with the contextual understanding of generative AI models. The project involved conducting a comprehensive experimental evaluation of several existing transpilers, assessing their execution time, CPU and memory usage, and translation accuracy. Building upon these findings, an LLM-enhanced transpilation framework was developed, introducing several innovative strategies: splitting ASTs into context-aware chunks to stay within model limitations, automatically generating unit tests to verify correctness of translated code, and enabling whole-project transpilation rather than single-file conversion. The final stage of the study compared the performance and precision of this hybrid approach against the benchmarked tools.

Keywords LLMs, Source-to-source translation, AST, Code transformation, Semantic analysis, Transpilation, Code generation

Resumo

Nos últimos anos, o rápido crescimento dos Modelos de Linguagem Grande (LLMs) transformou vários domínios, inspirando novas possibilidades para automatizar tarefas cognitivas complexas — incluindo engenharia de software. Este trabalho explora o potencial de aproveitar os LLMs para acelerar a migração de bases de código legado para ambientes modernos. Os transpilers tradicionais, que convertem código-fonte de uma linguagem para outra, há muito dependem de abordagens baseadas em regras e transformações Abstract Syntax Tree (AST). Embora esses métodos sejam confiáveis para conversões sintáticas, eles frequentemente enfrentam dificuldades com construções sensíveis ao contexto, lógica específica da estrutura e manutenção da fidelidade semântica entre paradigmas de programação distintos. Para enfrentar esses desafios, esta pesquisa integra técnicas de aprendizagem automática a uma arquitetura de transpilador contemporânea, criando um novo sistema híbrido que combina a precisão estrutural da transpilagem baseada em AST com a compreensão contextual dos modelos generativos AI. O projeto envolveu a realização de uma avaliação experimental abrangente de vários transpiladores existentes, avaliando o seu tempo de execução, uso de CPU e memória e precisão de tradução. Com base nessas descobertas, foi desenvolvida uma estrutura de transpilagem aprimorada por LLMs, introduzindo várias estratégias inovadoras: dividir ASTs em blocos sensíveis ao contexto para permanecer dentro das limitações do modelo, gerar automaticamente testes de unidade para verificar a correção do código traduzido e permitir a transpilagem de todo o projeto em vez da conversão de um único ficheiro. A fase final do estudo comparou o desempenho e a precisão dessa abordagem híbrida com as ferramentas de referência.

Palavras-chave Linguagem de Grande Porte (LLM), Tradução de fonte para fonte, Árvore sintáctica abstrata (AST), Transformação de código, Transpilação, Geração de código

Contents

List of Figures

Acronyms

AI Artificial Intelligence.

ANTLR Another Tool for Language Recognition.

APIs Application Programming Interface.

AST Abstract Syntax Tree.

CLI Comand Line Interface.

CPU Central Processing Unit.

DSL Domain-Specific Languages.

DSR Design Science Research.

DTOs Data Transfer Objects.

GLSL OpenGL Shading Language.

GPT Generative Pre-trained Transformer.

HLSL High-Level Shader Language.

JSON JavaScript Object Notation.

LLMs Large Language Models.

LSTM Long-Short-Term Memory.

ML Machine Learning.

NLP Natural Language Processing.

ORM	Object-relational Mapping.
RL	Reinforcement Learning.
UML	Unified Modeling Language.

Chapter 1

Introduction

Code translation represents a critical challenge in software engineering, since it involves the transformation of the source code from one programming language to another while preserving its fundamental functionality. Traditionally, developers have relied on transpilers–specialized programs designed to automate this complex translation process. However, the development of a comprehensive transpiler is inherently challenging and resource intensive (Plaisted, 2013).

The landscape of software development is rapidly evolving, with Artificial Intelligence (AI) increasingly integrating into the various domains of computational engineering. Recent technological advancements have expanded the potential applications of machine learning techniques in software development processes, particularly in areas such as:

- Vulnerability detection in software systems;
- Sophisticated program analysis techniques;
- Automated code generation and transformation.

Recent breakthroughs in LLMs have raised significant interest in exploring alternative code translation methodologies. These advanced computational models present a potential paradigm shift in the way we approach source code transformation, challenging traditional transpilation techniques (Pettorossi et al., 2024).

This research aims to conduct a comprehensive comparative analysis of transpilers and models in the large language domain of code translation. The primary focus is to empirically evaluate and compare these approaches, examining their performance in translating source code from legacy programming languages to modern programming environments. By investigating the nuanced capabilities of both transpilers and

LLMs, it is desired to provide insights into their respective strengths, limitations, and potential synergies in software engineering processes.

Looking into future implementations, on the transpiler side it will be used Another Tool for Language Recognition (ANTLR), a sophisticated parser generator that enables precise text input processing based on predefined grammatical rules. This tool will serve as a foundational framework for constructing a robust source-to-source compiler and facilitating the desired comparative analysis. On LLMs side, there is still research to be done on the next phases and a decision should be taken then.

1.1 Theoretical Context

The domain of code translation has witnessed the appearance of potential transformative advancements through machine learning techniques, with transformer models emerging as a particularly promising approach. These models represent a paradigm shift in understanding and generating programming language constructs, leveraging sophisticated neural network architectures.

Transformer Architecture in Code Translation

Transformer models, originally introduced by (Vaswani et al., 2023), have demonstrated remarkable capabilities in natural language processing and have been increasingly adapted to programming language translation. Key characteristics that distinguish these models include:

- Self-attention mechanisms enabling contextual understanding of code semantics;
- Parallel processing of code tokens, enhancing computational efficiency;
- Ability to capture complex syntactical and semantic relationships across programming languages.

Machine Learning Techniques for Code Representation

Several prominent approaches have emerged in representing and translating code using machine learning:

1. **Sequence-to-Sequence Models**: Neural network architectures that map input code sequences to equivalent output sequences, focusing on preserving functional semantics Kim (2021).

2. **Pre-trained language models**: Large-scale models such as CodeBERT (Feng et al., 2020) and Generative Pre-trained Transformer (GPT)-based variants (Radford and Narasimhan, 2018) that leverage extensive code corpora to generate contextually aware translations.

3. **Representation Learning**: Techniques that transform code into dense vector representations, enabling a more nuanced understanding of programming language structures (Bengio et al., 2013).

Challenges in Machine Learning-Based Code Translation

Despite everything mentioned above, several critical challenges still persist:

- Maintaining precise semantic equivalence during translation;
- Handling language-specific idioms and programming paradigms;
- Generating human-readable and optimized target code;
- Addressing the computational complexity of large-scale translation tasks.

Koehn and Knowles (2017)

1.2 Motivation

The increasing complexity of software systems and the rapid evolution of programming languages create significant challenges for organizations to maintain and modernize legacy codebases. Traditional code translation methods have proven to be time-consuming, error-prone, and resource-intensive, so there is room for innovative approaches that can streamline the transformation process while maintaining code integrity and functionality (Pettorossi et al., 2024).

Machine learning techniques, particularly transformer-based models, present a promising alternative to conventional transpilation methods (Vaswani et al., 2023). These advanced computational approaches offer the potential to automate and optimize code translation processes, addressing critical limitations in existing software modernization strategies. By leveraging sophisticated neural network architectures, machine learning models can potentially decode complex programming language semantics more effectively than traditional rule-based transpilers (Raina et al., 2024).

The motivation for this research comes from two primary considerations:

1. First, the growing technical debt accumulated in legacy systems poses substantial operational risks for organizations. Many enterprises continue to rely on outdated programming languages that become increasingly difficult to maintain, scale, and secure. Efficient code translation mechanisms can help mitigate these risks by enabling smoother technological transitions.

2. Second, the computational linguistics underlying machine learning models has demonstrated remarkable capabilities in understanding and generating human-like text. Extending these capabilities to programming languages represents a potential a major technological innovation. The ability to accurately translate code while preserving semantic nuances could revolutionize software maintenance and modernization processes.

Contributions to Industry and Development Ecosystem

The outcomes of this Master's project will assist developers and organizations by offering several strategic advantages:

1. **Cost Reduction:** By providing a comparative analysis of machine learning and traditional transpilation approaches, research can help organizations make more informed investment decisions in technology. The potential for reducing manual translation efforts can produce into significant cost savings.

2. **Technical Risk Mitigation:** The study will offer empirical information on the reliability and precision of machine learning-based code translation, helping organizations assess the feasibility of adopting these emerging technologies in their software modernization strategies.

3. **Optimization Strategies:** The research will provide detailed performance metrics and comparative analysis, enabling developers to understand the strengths and limitations of different code translation methodologies.

4. **Innovation Acceleration:** The findings can inspire further research and development in machine learning applications for software engineering, potentially catalyzing more advanced translation and transformation techniques.

For small to medium enterprises struggling with technological legacy systems, this research offers a potential pathway to more agile and adaptable software infrastructure. Large technology organizations can leverage these insights to develop more sophisticated code transformation strategies, reducing the friction associated with technological transitions.

Ultimately, the presented work aims to bridge the gap between machine learning techniques and practical software engineering challenges, providing a rigorous exploration of how artificial intelligence can transform the code translation processes.

1.3 Research Objectives

This research aims to explore the integration of machine learning techniques with traditional AST-based transpilers to improve the accuracy and efficiency of cross-language code translation. Using the strengths of machine learning models to handle complex and context-dependent translation.

To address these challenges, this research proposes a hybrid approach that combines traditional AST-based transpilation with modern machine learning techniques, which has been confirmed to have significant performance gains in the execution of deep learning code (Tavarageri et al., 2021). After that context-sensitive translation mechanisms will be implemented and can preserve program semantics across different programming paradigms (Lachaux et al., 2021). In the end, it is intended to evaluate the system's performance against traditional transpilers using established code quality metrics.

This research addresses a critical challenge in software engineering: the efficient and accurate translation of source code between programming languages. As technological landscapes evolve, organizations face increasing pressure to modernize legacy systems while maintaining code functionality and semantic integrity.

The purpose of this study is to compare Large Language Models (LLMs) to conventional transpilers in a number of important areas. First, it looks at how well they perform in terms of handling language-specific constructs, translation accuracy, and semantic preservation. It does this in an effort to determine whether LLMs can continue to provide the same degree of accuracy and dependability as rule-based AST conversions have historically been able to. The research's consideration of the development effort required for each strategy is another crucial component. The paper examines the differences between using LLMs for code translation and developing and maintaining a transpiler, taking into account both approaches' resource and computational requirements in addition to human labour. Additionally, the study assesses the viability of implementing LLM-based solutions for actual code migration activities. This entails determining whether building a specialised transpiler or using LLMs as a translation engine is more beneficial given trade-offs between accuracy and development effort. Lastly, the study looks into how much semantic equivalency LLMs can maintain while translating. It looks at how well these models represent the original code's intent and structural meaning, particularly when working with intricate programming paradigms or idioms unique to a given language that make conventional transpilation methods difficult to use.

Research Boundaries

While acknowledging the breadth of potential investigations, this study will focus specifically on:

- Code translations between selected programming languages;
- Comparative analysis of transpilers and LLMs approaches;
- Performance evaluation using predefined accuracy and efficiency metrics.

1.4 Technical Challenges and Innovation

The integration of machine learning techniques into code translation represents a complex computational challenge across multiple domains of software engineering and artificial intelligence. This work faces technical obstacles that extend beyond syntactical transformations, diving into the domain of semantic preservation and code understanding.

Applying machine learning to transpilation demands sophisticated approaches to manage contextual complexity. Traditional translation methods are more prone to errors when facing the challenge of translating between different programming paradigms, whereas advanced neural network architectures offer promising alternatives. **Transformer-based models**, particularly those that use self-attention mechanisms, demonstrate remarkable capabilities in decoding more complex code structures in different programming languages.

The primary architectural challenge emerges from the need to develop models that can comprehend not just the literal syntax, but the underlying computational intent (Bastidas Fuertes et al., 2023). This requires sophisticated sequence-to-sequence neural networks capable of capturing implicit programming conventions and handling language-specific idioms with precision. Hybrid architectures that combine neural network approaches with rule-based systems present particularly intriguing possibilities for addressing these complex translation demands (Lachaux et al., 2021).

Performance evaluation becomes a multidimensional challenge, necessitating a comprehensive assessment framework that extends beyond traditional metrics. Computational performance metrics such as execution time, Central Processing Unit (CPU) consumption, memory utilization, and energy expenditure provide critical information on the practical viability of machine learning-based translation approaches. Simultaneously, translation quality must be rigorously assessed through metrics that evaluate semantic accuracy, functional equivalence, and compilation success rates.

The validation strategy demands meticulous design, by incorporating standardized test suites across multiple programming languages and implementing controlled experimental environments. Statistical analysis becomes crucial in understanding the performance variations and potential limitations of machine learning translation models.

Potential limitations in this domain are significant. Machine learning models must demonstrate robust capabilities in handling complex, domain-specific code structures, managing translations of legacy programming paradigms, and mitigating inherent model biases. These challenges require sophisticated mitigation strategies that blend advanced machine learning techniques with deep computational linguistics understanding (Le et al., 2020). As well has creating a framework able to test and validate to a significant extent the produced outputs.

By systematically addressing these technical complexities, this work aims to improve the automated code transformation processes, offering a rigorous, empirical exploration of how artificial intelligence can improve software translation processes.

1.5 Methodological Approach

The structural design ensures a rigorous, systematic approach to investigating machine learning techniques in code translation:

- Establishing theoretical context
- Reviewing existing methodologies

- Conducting empirical investigation
- Critically analyzing results

To accomplish this Master Thesis objectives, it is used this iterative methodology based on literature revision, solution proposal,implementation and testing.

To carry out this approach, the working plan is composed of the following six steps that follow Design Science Research (DSR) phases (Peffers et al.) :

- Identify the problem and the motivation to solve it:
  - Bibliographic study to deeply understand transpilers, their current architectures, and fundamental limitations;
  - Bibliographic study to understand ML and Artificial Intelligence (AI) concepts, specifically their application in transpiler optimization.
- Define objectives for the solution:
  - Establish quantitative and qualitative metrics for transpiler enhancement evaluation;
  - Specify functional and non-functional requirements of the proposed approach.
- Design and Development :
  - Investigation and development of ML models for code transformation optimization;
  - Development of intermediate code representation suitable for neural network processing;
  - Implementation of inference mechanisms for optimized code generation.
- Demonstration :
  - Application of developed models on representative test datasets;
  - Execution of comparative experiments between traditional and ML-based approaches;
  - Analysis of optimization impact on generated code quality.
- Evaluation:
  - Quantitative analysis of model performance and efficiency;

    - Qualitative assessment of code transformation adequacy;
    - Validation through real-world use cases;
    - Identification of limitations and future improvement opportunities.
- Communication:
    - Complete documentation of methodology and obtained results;
    - Publication in relevant scientific venues;
    - Open-source release of implementation and datasets.

This methodology ensures systematic development of a solution that enhances transpilation through ML techniques, advancing the state of the art in compiler technology.

1.6 Document Structure

This document is divided into seven chapters; the first one is the introduction where important topics like context, motivation, objectives, challenges, and approaches are discussed. The introduction highlights the importance of code translation in software engineering, especially with the rapid evolution of technology and the need to modernize legacy systems. It emphasizes that traditional transpilation methods often struggle with preserving semantic integrity and handling language-specific constructs, which are critical for maintaining code functionality post-translation.

Chapter 2 is the state-of-art study about transpilers where a lot of relevant information about the transpilation process and transpilers tools are explained. The study reported includes the exploration of advanced transpilation technologies and their critical role in modern software development. Transpilers bridge language ecosystems, enabling developers to use cutting-edge features while maintaining compatibility with existing environments. They facilitate language interoperability, feature adoption through polyfills, and streamline technological migration and codebase modernization. It also will explain how transpilers support cross-language development and their respective testing. However, challenges such as semantic preservation, performance overheads, and debugging complexity must be addressed.

Chapter 3 explores the integration of Artificial Intelligence (AI) into code translation, primarily through transpilers (source-to-source compilers), addressing the limitations of traditional rule-based systems which struggle with complex language features and preserving functionality. AI simulates human intelligence processes, such as learning and reasoning, making transpilation more efficient and adaptable. Machine Learning is central, enabling systems to learn syntax and semantics from code repositories, predict optimal transformations, and infer code intent. ML employs paradigms like supervised learning for accurate code mapping, unsupervised learning for structural analysis, and reinforcement learning for optimizing translation strategies based on performance feedback. Concurrently, Natural Language Processing (NLP) treats programming languages as structured languages, utilizing techniques such as tokenization, parsing, and semantic analysis to ensure deep code understanding and precise transformations. Beyond basic translation, AI extends transpiler capabilities to include Automated Refactoring for code quality improvement, sophisticated Error Detection and Correction via static and dynamic analysis, and advanced Cross-Linguistic Translation models (like sequence-to-sequence transformers) that tackle syntactic divergence and guarantee semantic preservation across different paradigms.

Chapter 4, the proposal, presents the system architecture designed for comparative analysis between machine learning-based approaches and traditional transpilation methods in code translation. The proposed framework should integrate multiple components to enable rigorous evaluation based on performance metrics, accuracy, and resource utilization.

Chapter 5 explores an independent comparative analysis of various code transpilers, selected largely based on their popularity and adoption by the developer community on **GitHub**. The study's methodology involved testing these tools using code extracts that all implemented the exact same functional logic—processing a list of numbers to ensure a consistent basis for comparison across different source languages *(like* **JavaScript/TypeScript, C**, *and* **Java***)*. Practical testing was challenging, as several initially selected tools could not be successfully installed on the Linux testing environment, restricting the study to a subset of operational transpilers. For those successfully tested, the core evaluation focused on four main metrics: execution time, CPU usage, memory consumption, and translation accuracy (or fidelity).

Chappter 6 detailed the architecture and implementation of an automated **Java-to-Python** translation system designed specifically to migrate legacy applications by moving beyond lexical substitution and operating entirely on the Abstract Syntax Tree (AST). Structured as a modular, multi-phase pipeline, the system first used the **Java Parser Frontend** (powered by **ANTLR**) to convert source code into a structured JSON AST. A critical Framework and Context Analyzer component then performed static analysis to detect the use of major technologies, including **Spring, Hibernate, and Struts**, compiling this evidence into metadata essential for architectural translation. The core translation logic resided in the AI Transformation Core, which leveraged Google's Gemini Large Language Model. The model was strictly governed by an extensive system prompt that encoded hundreds of explicit rules for sophisticated AST-to-AST mapping, enabling framework migration *(e.g., from* **Spring MVC** *to* **Flask***)* and the translation of core language constructs. To ensure reliability with large enterprise files, a structure-aware chunking strategy was implemented. Finally, the **Python**AST Reconstructor and Code Generator took the AI's translated JSON, validated and merged the resulting **Python** AST fragments, and reliably generated the final, executable **Python** source code using the native `ast.unparse()` function.

Chapter 7 is the conclusion, and provides a systematic synthesis of the document, moving from theoretical foundations to empirical investigation and final insights. Also it asks if that all initially proposed objectives were successfully achieved, so it can validate that the integration of Machine Learning techniques significantly enhances the overall effectiveness of transpilation processes. Future work was recommended to address minor drawbacks and potentially extending the methodology to legacy programming languages like COBOL.

Chapter 2

Transpilers - Bibliographic Review

As software development continues to evolve, transpilers have become essential tools to convert source code between different programming languages and to manage compatibility between multiple language versions. This capability is particularly important in projects that need to maintain code across different platforms or need to gradually migrate between programming languages while preserving functionality.

2.1 Defining Transpilers

Transpilers, also known as source-to-source compilers, are specialized tools that enable a unique form of code transformation by translating source code from one programming language or version to another while preserving its fundamental computational semantics (Fuertes et al., 2023). Unlike traditional compilers that convert source code directly to machine code, transpilers operate at a higher level of abstraction, facilitating complex language translations that maintain the original code's intent and structure (Schneider and Schultes, 2022).

The conceptual framework of transpilers is distinguished by a three-stage transformational process: parsing the source language into an Abstract Syntax Tree, performing semantic and syntactic transformations, and generating equivalent code in the target language (Ilyushin and Namiot, 2016). This approach provides developers with a clear method that intends to give flexibility and reach the final objective: to translate code from a source to a target language.

Nicolini et al. (2023) emphasizes that transpilers are critical in addressing the dynamic nature of programming languages, enabling developers to adopt modern language features while ensuring backward compatibility (in terms of functionality) and system interoperability. A widely known example is the TypeScript transpiler, which allows developers to use advanced JavaScript features that are compiled down to earlier *ECMAScript* versions, thus ensuring broader browser and platform support (Japikse et al., 2019).

The significance of transpilers extends beyond mere code translation. It serves as a crucial tool for software modernization, enabling technological migration and providing developers with mechanisms to overcome compatibility limitations inherent in traditional software development approaches.

2.2 Source-to-Source Translation

The development of source-to-source translation techniques represents a critical trajectory in the evolution of programming language technologies, reflecting the ongoing challenges of software adaptation and language interoperability. This section explores the foundational origins and subsequent technological progression that have shaped transpilation methodologies.

Source-to-source translation emerged from the need to overcome limitations in programming language design and compatibility. (Pettorossi et al., 2024) Early computational research in the 1960s and 1970s recognized the potential for program transformation as a mean to address technological constraints and expand the capabilities of emerging programming paradigms. Researchers initially explored program transformation techniques as a way to optimize code, improve portability, and enable more sophisticated language abstractions (Gaudillière-Jami, 2020).

The earliest forms of source-to-source translation were primarily focused on program optimization and platform-specific adaptations. Fortran and COBOL preprocessors from the 1960s can be considered precursors to modern transpilers, demonstrating the initial conceptual approaches to source code transformation. These early systems attempted to translate code between different dialects or add preprocessor-level capabilities that extended the expressiveness of existing programming languages (Johnson, 1987) (Allen, 2024).

2.3 Technological Progression

The evolution of source-to-source translation witnessed significant milestones with the surge of more sophisticated programming paradigms and the increasing complexity of software systems. The 1980s and 1990s saw the development of more advanced program transformation techniques, particularly in functional and object-oriented programming languages (Berdonosov and Zhivotova, 2014). Researchers began developing more sophisticated tools that could perform complex semantic-preserving transformations across different programming language abstractions.

Key technological advancements included the development of more robust parsing techniques, the introduction of AST representations, and the creation of increasingly sophisticated transformation algorithms. The rise of multi-paradigm programming languages and the need for cross-language interoperability further accelerated the development of more advanced transpilation techniques (Grune and Jacobs, 2008).

The emergence of web technologies in the late 1990s and early 2000s marked a new branch in the transpilers ecosystem. JavaScript transpilers, such as Babel [1], became crucial in addressing the rapid evolution of web standards and browser compatibility challenges. These tools enabled developers to use modern language features while maintaining support for older browser environments, effectively bridging technological generations. (Andrés and Pérez, 2017)

2.4 Parsing Methodologies

Parsing serves as the initial stage of a transpilation process, transforming raw source code into a structured representation that can be analyzed and manipulated. The parsing phase involves two critical components: lexical and syntactic analysis (Cox and Clarke, 2003).

Lexical analysis breaks down the source code into a sequence of tokens, identifying individual language constructs such as keywords, identifiers, literals, and operators. Syntactic analysis then examines these tokens to verify their compliance to the defined language's grammatical rules and constructs a hierarchical representation of the code.

Different parsing techniques have been developed to handle the complexities of modern programming languages. These include top-down parsing methods like recursive descent parsing, bottom-up approaches such as LR parsing, and more advanced techniques like GLR (Generalized LR) parsing that can handle ambiguous and context-sensitive grammatical structures (Goodman, 1999) (Johnstone et al., 2006). Each methodology offers advantages and disadvantages in terms of performance, flexibility, and complexity of language parsing.

[1] https://babeljs.io/docs/

2.5 Abstract Syntax Tree

The Abstract Syntax Tree represents a representation that captures the structural and semantic essence of the source code. An AST is a tree-like data structure where each node represents a construct in the source code, abstracting away syntactic details while preserving the essential computational meaning (Jiang et al., 2021). This representation provides a structured intermediate form that abstracts the syntactic details of source code into a tree-based data structure. While AST enable code analysis and transformation, the complete transpilation process typically requires multiple steps of transformation between the source AST and target code generation, including semantic analysis and language-specific optimizations (Raina et al., 2024).

Key characteristics of AST include:

- Hierarchical representation of code structure;
- Removal of syntactic sugar and redundant grammatical elements;
- Preservation of semantic relationships between code constructs;
- Facilitation of language-independent code transformations.

(Grune and Jacobs, 2008)

The generation of an AST involves traversing the parse tree and changing it to reflect the grammatical and semantic relationships within the original source code. Each node in the tree can represent various programming constructs such as function declarations, control flow structures, variable assignments, and expressions (Grosser et al., 2015).

2.6 Code Transformation

Code transformation represents the core computational process of transpilation, where the AST is systematically modified to generate equivalent code in the target language or version (Thongtanunam et al., 2022). This phase involves multiple strategies to ensure semantic preservation while adapting to the target language's specific syntactic and structural requirements (Baar and Marković, 2007).

The transformation process typically encompasses **several key strategies:**

- Structural transformation of language constructs;

- Semantic mapping between different language features;
- Handling of language-specific idioms and patterns;
- Generating optimized and compatible target code.

Complex transformations may involve multiple traversals over the AST, each one addressing different aspects of code translation (Thongtanunam et al., 2022). These can include:

- **Feature adaptation** (e.g., translating modern JavaScript features to ES5);
- **Paradigm translation** (converting functional programming constructs to imperative styles);
- **Semantic equivalence preservation**;
- **Performance optimization**.

2.7 Typology of Transpilers

The diverse landscape of transpilation technologies can be categorized into distinct typological classifications, each addressing specific challenges in software development and language interoperability. This section explores the primary taxonomies of transpilers, highlighting their unique characteristics and technological applications.

2.7.1 Technical Mechanics of Transpilation

Transpilation represents a complex process of source code transformation that requires sophisticated computational techniques to ensure accurate and semantically preserving translation between programming languages or language versions. This section delves into the intricate technical mechanisms that underpin the transpilation process.

2.7.2 Language-to-Language Transpilers

Language-to-language transpilers represent an approach to cross-language code transformation, enabling developers to translate source code between fundamentally different programming languages. These transpilers address the challenge of linguistic heterogeneity in software ecosystems, facilitating code reuse, platform migration, and technological integration (Wang and Wang, 2024).

The translation process involves semantic mapping, addressing not just syntactic differences but also paradigmatic variations between source and target languages. For example, transpilers might translate between statically-typed and dynamically-typed languages, or between languages with different memory management approaches, requiring sophisticated computational strategies to preserve the original code's computational intent (Plaisted, 2013) (Kratchanov and Ergün, 2018).

2.7.3 Version Transpilers

Version transpilers focus on managing the evolutionary challenges of programming languages by enabling code migration between different versions of the same language. These tools are particularly useful in rapidly evolving language ecosystems where backward compatibility and feature adoption present a significant challenge for developers (Bastidas Fuertes et al., 2023).

Modern version transpilation goes beyond syntax updates, addressing semantic changes, deprecation of language features, and introducing modern language capabilities to legacy codebases. JavaScript transpilers like Babel exemplify this approach, allowing developers to use next-generation ECMAScript features while maintaining compatibility with older browser environments (Thomas Schoenemann, 1999).

2.7.4 Paradigm Transpilers

Paradigm transpilers represent the most complex form of source code transformation, addressing the fundamental challenges of translating between different programming paradigms. These advanced transpilation techniques attempt to bridge conceptual differences between computational approaches such as functional, imperative, object-oriented, and declarative programming models. The translation between programming paradigms, requires deep understanding of computational semantics, involving complicated transformation strategies that go beyond syntactic conversion (Andrés and Pérez, 2017).

2.8 Web Development Ecosystem

The web development landscape has been particularly revolutionized by transpilation technologies, addressing the intricate challenges of browser compatibility, language evolution, and feature adoption. JavaScript transpilers have become instrumental in enabling developers to leverage modern language features while maintaining broad platform support.

Tools like Babel have transformed web development by allowing developers to write code using the latest ECMAScript specifications while generating compatible code for older browser environments. This approach enables immediate adoption of cutting-edge language features, such as arrow functions, destructuring, async/await, and module systems, without sacrificing compatibility with legacy systems (Nicolini et al., 2024).

Beyond JavaScript, transpilers have facilitated the integration of alternative language technologies into web development. TypeScript and CoffeeScript exemplify how transpilation enables developers to use enhanced type systems and alternative language syntaxes while ultimately generating standard web-compatible JavaScript Maharry (2013).

2.9 Software Modernization

Software modernization represents a major application of transpilation technologies, addressing the challenges of legacy system evolution and technological migration. Transpilers provide a strategic approach to gradually updating and transforming existing codebases, enabling organizations to incrementally adopt modern programming practices without complete system rewrites.

The modernization process involves complex transformations that go beyond syntax updates. **Transpilers can facilitate:**

- Migration between programming language versions;
- Adaptation of legacy code to modern architectural patterns;
- Integration of contemporary programming paradigms;
- Gradual technological stack updates.

By providing a mechanism for controlled and predictable code transformation, transpilers mitigate the risks associated with large-scale system rewrites, offering a more economical and less disruptive approach to technological evolution (Neumann, 1996).

2.10 What problem does transpilers solve

Transpilers address several fundamental challenges in software development and language engineering (Fuertes et al., 2023):

Compatibility Barriers

Transpilers overcome the limitations imposed by various technological ecosystems, allowing code to run on different platforms, browsers, and runtime environments. They bridge technological gaps that would otherwise prevent code reuse and cross-platform development.

Language Feature Adoption

By providing a mechanism for backward compatibility, transpilers allow developers to immediately leverage advanced language features without waiting for complete environment support. This accelerates the adoption of innovative programming techniques and language improvements.

Technical Debt Management

Organizations can use transpilers as a strategic tool for managing technical debt, gradually modernizing codebases without requiring complete rewrites. This approach allows incremental improvements and reduces the risks associated with massive system transformations.

Paradigm and Architectural Flexibility

Transpilers enable more flexible approaches to software design by facilitating translations between different programming paradigms and architectural styles. This flexibility supports more adaptive and innovative software development strategies.

Performance and Optimization

Advanced transpilation techniques can introduce performance optimizations, transforming code to more efficient representations while maintaining the original computational intent.

Code Maintainability

By using transpilers, developers can write cleaner, more maintainable code in languages or dialects that offer better syntax, type checking, or other enhancements. This code is then transpiled into a standard language for execution.

Cross-Language Integration

Transpilers facilitate the integration of different programming languages within a project. For example, TypeScript and CoffeeScript transpile to JavaScript, allowing developers to use these languages' features while still producing standard JavaScript code.

2.11 Performance Considerations

Performance represents a critical concern in transpilation technologies, introducing computational overhead that can potentially impact software efficiency. The process of source-to-source translation inherently involves multiple stages of code analysis, transformation, and regeneration, each of which contributes to computational complexity and potential performance degradation (Fang et al., 2023).

The performance challenges manifest in several dimensions. Firstly, the parsing and AST generation processes might require substantial computational resources, introducing additional processing time compared to direct compilation or interpretation. Each transformation adds layers of computational complexity, potentially increasing memory consumption and processing time (Rabinovich et al., 2017).

Moreover, the generated code might not always achieve the same level of optimization as hand-written or directly compiled code. Transpilers must balance between preserving semantic accuracy and generating performant code, a challenging trade-off that can result in suboptimal runtime characteristics. The additional abstraction layer introduced by transpilation can lead to performance penalties, especially in computationally intensive applications (Searcey, 2023).

2.12 Technical Complexities

The technical complexities of transpilation extend far beyond simple code translation, representing a domain of computational problem-solving. Maintaining semantic equivalence between source and target languages, requires a deep understanding of computational models, language-specific nuances, and intricate transformation strategies. Semantic preservation emerges as a challenge, demanding intricate mapping between different language constructs, type systems, and computational paradigms (Jin, 2024).

Transpilers must translate accurately not just syntactic structures but also preserve the fundamental computational intent across potentially vastly different language ecosystems.

The complexity is further amplified by the diversity of modern programming languages, each with unique type systems, memory management approaches, and computational abstractions. Translating between languages with fundamentally different paradigms—such as from a functional to an imperative programming model—requires sophisticated algorithmic approaches that can accurately represent complex computational behaviors Alves et al. (2012).

One of the most challenging aspects of transpilation involves translating language-specific instructions that lack direct equivalents in the target language. Consider the translation from C to Python, which illustrates the intricate nature of semantic preservation. In C, low-level memory management and pointer manipulation represent critical areas where direct translation becomes problematic.

Consider this C function that demonstrates direct memory manipulation and pointer arithmetic that cannot be straightforwardly translated to Python:

```
void modify_memory_directly(int* buffer, size_t size) {
    for (size_t i = 0; i < size; i++) {
        ((char)buffer + i) = (((char)buffer + i) + 1) % 256;
    }
}
```

(MinéAntoine, 2006)

Translating this C function to Python presents several fundamental challenges that highlight the inherent differences in memory management and low-level system interactions between the two languages:

- **Pointer Manipulation Limitations**: Python does not support direct pointer arithmetic or type casting in the same manner as C. The explicit conversion of an integer pointer to a character pointer and subsequent byte-level manipulation is not possible in Python's memory model.
- **Memory Access Restrictions**: The function performs direct byte-level memory modification, which is categorically prevented in Python. Python's memory management abstracts away low-level memory access, prioritizing safety and type integrity over direct memory manipulation.

- **Type Safety and Abstraction**: Python's type system and memory management prevent the kind of unsafe, low-level operations demonstrated in the C code. The ability to directly increment bytes of an integer buffer is fundamentally incompatible with Python's memory abstraction.
- **Computational Semantics**: The modulo operation on individual bytes, effectively creating a cyclic byte increment, cannot be directly replicated in Python without significant restructuring of the algorithm.

A theoretical Python translation would require a completely different approach, potentially using byte arrays or alternative algorithms that respect Python's memory management constraints. This example illustrates the non-trivial challenges in translating low-level system programming constructs between languages with fundamentally different memory models and design philosophies.

As Bastidas Fuertes et al. (2023) proved debugging transpiled code presents another significant technical challenge. The generated code often diverges substantially from the original source, making traditional debugging techniques less effective. Developers must navigate the additional complexity of understanding transformations and tracing issues through multiple layers of code generation .

2.13 Summary

The journey of transpilation technologies represents a pivotal narrative in the evolution of software development, reflecting the dynamic nature of programming languages and the continuous quest for more flexible, adaptive computational approaches.

The contemporary landscape of transpilation technologies demonstrates a remarkable maturity and sophistication. Modern transpilers have evolved from simple code translation tools to complex systems that enable fundamental transformations across programming paradigms, languages, and technological ecosystems.

Current transpilation technologies excel at addressing critical challenges in software development, including language interoperability, feature adoption, and technological migration. Tools like *Babel* for *JavaScript*, *TypeScript* transpilers, and cross-language transformation frameworks have become integral components of contemporary software engineering workflows. These technologies provide developers with unprecedented flexibility in managing the complexity of evolving programming languages and platforms (Fuertes et al., 2023).

The future of transpilation technologies presents exciting opportunities for further research and technological innovation. Emerging research directions suggest several promising avenues of exploration:

- Artificial intelligence and machine learning techniques are increasingly being integrated into transpilation processes. These approaches promise more intelligent, context-aware code transformations that can potentially generate more optimized and semantically precise translations across programming languages (Melo et al., 2023).
- There is growing interest in developing more sophisticated transpilers that can handle increasingly complex paradigm shifts. Research is focusing on creating more nuanced transformation techniques that can accurately translate between fundamentally different computational models, such as functional, imperative, and declarative programming paradigms (Lyu et al., 2023).

Additionally, the growing complexity of distributed and cloud-native computing environments demands more advanced transpilation techniques. Future research will likely explore how transpilers can facilitate more integration across diverse technological ecosystems, supporting increasingly complex software architectures (Ringlein et al., 2020).

The ongoing evolution of transpilation technologies reflects a broader trend in software engineering: the pursuit of more flexible, adaptable, and intelligent computational tools. As programming languages continue to evolve and new technological challenges emerge, transpilers can play a crucial role in bridging technological boundaries and enabling more innovative software development approaches.

Chapter 3

AI in Code Translation - Bibliographic Review

Artificial Intelligence refers to the simulation of human intelligence processes by machines, particularly computer systems. These processes include learning (the ability to improve from experience), reasoning (drawing conclusions from data), problem-solving, and understanding language. In the realm of software development, AI encompasses various technologies that enhance automation, decision-making, and user interaction.

Key components include Machine Learning (ML), which enables algorithms to learn from data and improve over time, thereby automating tasks and enhancing user experiences (Bennaceur and Meinke, 2018). Natural Language Processing (NLP) facilitates human-computer interaction by allowing systems to understand and generate human language, essential for applications like chatbots and virtual assistants (Joshi, 1991). Deep learning, a subset of ML, utilizes neural networks to recognize patterns and make predictions, driving advances in complex AI models (Salakhutdinov, 2014). Furthermore, computer vision empowers applications to interpret visual data, crucial for technologies such as self-driving cars (Freeman, 1999). Together, these technologies define the scope of AI in software development and transform the way applications are built and operated.

Among the various tools in software development, transpilers, short for ”source-to-source compilers”, hold a significant place. A transpiler converts source code from one programming language to another while maintaining a similar level of abstraction. Transpilers are essential for tasks such as:

- Migrating legacy code to modern programming languages.
- Enabling cross-platform compatibility by translating code for different environments.
- Optimizing code for performance or readability.

However, traditional transpilers often rely on predefined rules and heuristics to perform translations. While effective for straightforward conversions, these rule-based systems face challenges with more complex transformations, such as:

- Handling language-specific idioms or features unique to certain programming languages.
- Optimizing code for specific hardware or runtime environments.
- Preserving the original code's intent and functionality across different language paradigms.

(Fuertes et al., 2023)

This is where AI, particularly Machine Learning and NLP, can significantly enhance transpiler functionality. When applied to transpilers, ML can enable the system to:

- **Learn from Code Repositories:** By analyzing vast amounts of code, ML models can understand the syntax and semantics of different programming languages, leading to more accurate and efficient translations.
- **Predict Optimal Transformations:** ML can identify code patterns and suggest optimizations, such as improving performance or reducing resource usage.
- **Understand Code Intent:** ML can infer the intended functionality of code, ensuring that translations preserve the original behavior.

(Sharma et al., 2021)

Natural Language Processing, traditionally used for processing human languages, can be adapted to parse and interpret programming languages. By leveraging NLP techniques such as tokenization, parsing, and semantic analysis, transpilers can better grasp the structure and meaning of code, facilitating more precise transformations (Mihalcea et al., 2006). Moreover, AI can extend the capabilities of transpilers beyond simple translation. For instance:

- **Automated Refactoring:** AI-driven systems can suggest or apply improvements to code structure, enhancing readability, maintainability, or performance without human intervention.
- **Error Detection and Correction:** AI can learn from past code errors and successes, allowing the transpiler to identify potential bugs or inefficiencies and propose fixes.

- **Cross-Linguistic Translation:** AI can address challenges in translating between disparate programming languages by understanding their unique features and idioms.

(Bastidas Fuertes et al., 2023)

In summary, integrating AI into transpilers addresses the limitations of traditional rule-based systems, offering more intelligent, adaptable, and efficient tools for code translation and optimization. This research aims to provide a comprehensive understanding of how AI can revolutionize transpilation, making it a more powerful tool in modern software development.

3.1 Machine Learning Fundamentals

As mentioned previously, Machine Learning (ML) is a critical branch of artificial intelligence that empowers systems to learn from data and enhance their capabilities over time without explicit programming. This section explores the core concepts and key algorithms of machine learning, which are foundational for applications such as improving transpilers. By understanding these principles, we can use ML to enhance the precision of code translation, optimize performance, and detect errors effectively (Bennaceur and Meinke, 2018).

3.1.1 Core Concepts

Machine learning encompasses three primary paradigms: **supervised learning**, **unsupervised learning**, and **reinforcement learning**. Each offers unique approaches and applications, particularly relevant to tasks like transpilation.

Supervised Learning

In supervised learning, a model is trained on a labeled dataset where each input is paired with its corresponding output. The model learns to predict outputs for new inputs by minimizing prediction errors. For transpilation, supervised learning can map code from one language to another. For instance, a model trained on *Java-to-Python* code pairs could predict *Python* equivalents for *Java* constructs, such as converting a **Java for loop** into a **Python range-based loop** (Courtney et al., 2020).

Unsupervised Learning

This approach processes unlabeled data to uncover hidden patterns or groupings without predefined outputs. In the context of transpilers, unsupervised learning can cluster similar code segments or detect recurring patterns across languages, aiding in structural analysis or anomaly detection. For example, it might identify common coding idioms that inform more robust translation rules (Hu et al., 2020).

Reinforcement Learning

Reinforcement learning trains an agent to make sequential decisions by interacting with an environment, guided by rewards or penalties. Though less common in transpilation, it could optimize the translation process itself. An agent might learn to select translation strategies that maximize code performance, such as prioritizing memory-efficient constructs based on runtime feedback (Wang et al., 2022b).

3.1.2 Algorithms

Several machine learning algorithms are particularly suited to enhancing transpilers by analyzing, predicting, and generating code transformations.

Decision Trees

Decision trees partition data into branches based on feature values, culminating in decisions at the leaves. Their interpretability makes them ideal for classifying code elements or predicting translation choices (Suthaharan, 2016). For example, a decision tree could decide whether a conditional statement in *C++* translates best to an **if block** or a **ternary operator in Python**, based on syntax and context.

Neural Networks

Neural networks, especially deep learning variants, excel at modeling complex, non-linear relationships in data through layered nodes. In transpilation, they can capture code semantics or perform sequence-to-sequence translations. Models like Transformers, widely used in natural language processing, could translate entire codebases by learning contextual relationships between source and target languages (Li et al., 2023).

Regression Models

Regression models predict continuous outcomes, such as performance metrics, from input features. In transpilation, they can estimate the efficiency of translated code (e.g., execution time or resource usage), enabling the transpiler to select optimal translations. A linear regression model might correlate the complexity of the code with run-time, guiding optimization decisions (Marandi and Khan, 2015).

Together, these core concepts and algorithms enable machine learning to revolutionize transpilation, making it more intelligent and adaptive. Supervised learning drives accurate translations, unsupervised learning reveals structural insights, and reinforcement learning refines processes, while algorithms like decision trees, neural networks, and regression models provide the tools to execute these tasks effectively.

3.2 Natural Language Processing (NLP)

Natural Language Processing (NLP), a field of artificial intelligence focused on the interaction between computers and human language, is increasingly being adapted to process and transform programming languages. In the context of transpilation NLP offers powerful techniques to enhance code understanding and transformation. This section explores the role of NLP in transpilation and examines specific techniques, including tokenization, parsing, semantic analysis, and code generation, that enable more accurate and efficient code translation (Mihalcea et al., 2006).

3.2.1 Role in Transpilation

Transpilation goes beyond simple syntactic conversion; it requires a deep understanding of a program's structure, intent, and behavior to produce equivalent code in a target language. Traditional transpilers often rely on rigid, rule-based systems that struggle with complex language features, idiomatic expressions, or subtle semantic differences (Fuertes et al., 2023). NLP provides a more flexible and intelligent approach by treating programming languages as structured forms of language that can be analyzed and transformed using linguistic methods.

By leveraging NLP, transpilers can achieve the following:

- **Improved Code Understanding:** NLP techniques break down code into meaningful components and analyze its syntactic and semantic structure, enabling a clearer grasp of its logic and purpose.

- **Accurate Transformation:** By interpreting the intent behind the code, NLP ensures that translations preserve functionality and adhere to the conventions of the target language.
- **Context-Aware Solutions:** NLP allows transpilers to generate code that is not only correct but also optimized and idiomatic, adapting to the nuances of different programming paradigms.

(Ben-Nun et al., 2018)

This linguistic perspective makes transpilation more robust, reducing errors and enhancing the ability to handle diverse and complex codebases.

3.2.2 Techniques

NLP employs a series of techniques that form a pipeline for processing and transforming code. While originally developed for natural languages, these methods can be adapted to programming languages, accounting for their stricter syntax and precise semantics. Below, we explore four key techniques: tokenization, parsing, semantic analysis, and code generation.

Tokenization

Tokenization is the process of splitting code into smaller units, or tokens, such as keywords, identifiers, operators, and literals. Much like breaking a sentence into words in natural language, tokenization in code identifies the fundamental building blocks for further analysis. For example, in the *Python* statement **if x > 5:**, tokenization would produce the tokens **if, x, >, 5,** and **:**. This step is essential in transpilation, as it provides a granular view of the source code, enabling accurate interpretation and transformation into the target language (Grefenstette, 1999).

Parsing

Parsing analyzes the sequence of tokens to determine the grammatical structure of the code, typically generating a parse tree or Abstract Syntax Tree (AST). This tree represents the hierarchical organization of the code based on the language's syntax rules. For instance, in the expression **a + b * c**, parsing recognizes that multiplication takes precedence over addition, structuring the tree accordingly. In transpilation, parsing ensures that the relationships between code elements are understood and preserved, maintaining the correct logic and flow in the translated output (Sippu and Soisalon-Soininen, 2012).

Semantic Analysis

Semantic analysis goes beyond syntax to uncover the meaning and intent of the code. It examines aspects such as variable scopes, data types, and function behaviors to ensure the code's functionality is fully understood. For example, it might determine whether a variable is local or global or whether a function call is recursive. This step is critical in transpilation, as different languages may implement semantics differently (e.g., pass-by-value vs. pass-by-reference). By using techniques like control flow analysis and data flow analysis, semantic analysis ensures that the translated code behaves equivalently to the original (Wilhelm et al., 2013).

Code Generation

Code generation is the final stage, where the analyzed code is transformed into the target programming language. In NLP, this is akin to generating text based on an understanding of meaning; in transpilation, it involves producing code that is both syntactically correct and semantically equivalent. Advanced NLP models, such as sequence-to-sequence transformers, can be trained on large datasets of code to predict optimal translations. For instance, a **JavaScript forEach loop** might be translated into a **Python for loop**, preserving logic and adhering to *Python's* conventions. This step leverages insights from earlier techniques to produce high-quality, context-aware code (Shin and Jaechang, 2021).

These techniques work together to create a comprehensive process for transpilation. Tokenization and parsing handle the structural aspects of code, while semantic analysis and code generation focus on meaning and output, adapting NLP's strengths to the unique challenges of programming languages.

3.3 Code Analysis and Optimization

In the context of enhancing transpilers with machine learning techniques, code analysis and optimization represent critical areas where artificial intelligence can significantly improve the transformation process. This section examines the methodologies and approaches for applying machine learning to code analysis and optimization tasks.

3.3.1 Static vs. Dynamic Analysis

Static Analysis

Static analysis involves examining code without executing it, focusing on the structure, syntax, and potential behavior of programs (Louridas, 2006). Machine learning techniques have revolutionized static analysis by enabling more sophisticated pattern recognition and prediction capabilities.

Traditional static analyzers rely on predefined rules and heuristics, which often fail to capture complex code relationships. Machine learning models, particularly those based on graph neural networks and transformer architectures, can represent code as ASTs or control flow graphs and learn from vast repositories of code to identify patterns that human experts might miss (Allamanis, 2022).

These models excel at tasks such as:

- Identifying potential bugs and vulnerabilities
- Predicting code properties and behaviors
- Detecting patterns that may benefit from optimization
- Understanding dependencies and relationships between code components

For transpilers specifically, static analysis enhanced by machine learning can better preserve semantic equivalence between source and target languages by developing deeper representations of code meaning (Allamanis, 2022).

Dynamic Analysis

Dynamic analysis examines program behavior during execution, collecting runtime information that static analysis cannot access. Machine learning approaches to dynamic analysis typically involve:

- Profiling code execution to identify performance bottlenecks
- Monitoring memory usage and resource allocation patterns
- Tracing execution paths to understand program flow
- Recording input-output relationships to infer program semantics

Machine learning models trained on execution traces can predict which code paths are most frequently taken, allowing transpilers to prioritize optimization efforts accordingly (Lee et al., 1997). Furthermore, reinforcement learning techniques can be employed to iteratively test and refine code transformations based on runtime performance metrics (Wang et al., 2022b).

The integration of static and dynamic analysis through machine learning creates a powerful synergy for transpilers, enabling them to make more informed decisions about code transformation strategies that preserve both correctness and performance.

3.3.2 Machine Learning in Code Optimization

Predictive Optimization Models

Machine learning can transform code optimization from a rule-based process to a predictive one. By training on pairs of unoptimized and expertly optimized code, models can learn to identify opportunities for improvement and predict which transformations will yield the best results (Kulkarni and Cavazos, 2012).

These predictive models are particularly valuable in cross-language transpilation, where direct application of optimization techniques from the source language may not be optimal in the target language. Neural network architectures, especially sequence-to-sequence models and transformers, have demonstrated remarkable ability to learn mappings between equivalent but differently optimized code segments (Xiao and Guo, 2013).

Optimization Search Space Exploration

The space of possible code optimizations is vast, making exhaustive search impractical. Machine learning approaches can intelligently navigate this space using techniques such as:

- Bayesian optimization to efficiently search for optimal transformations
- Genetic algorithms that evolve code transformations toward better performance
- Deep reinforcement learning to learn optimization strategies through trial and error
- Meta-learning to transfer optimization knowledge across different code contexts

These approaches allow transpilers to discover novel optimization patterns that may not be captured by traditional compiler heuristics (Turner et al., 2021) (Cooper et al., 1999) (Wang et al., 2022a).

Context-Sensitive Optimization

Machine learning enables transpilers to develop context-sensitive optimization strategies that consider not just local code properties but broader application contexts. By analyzing patterns across entire codebases, ML models can make optimization decisions that account for:

- Target hardware characteristics
- Expected input distributions
- Interaction patterns between components
- Trade-offs between different optimization goals (e.g., speed vs. memory usage)

This holistic approach allows transpilers to generate code that is not just locally optimal but globally efficient within its deployment context (Yang et al., 2023).

3.3.3 Integration Challenges

Despite the promising capabilities of machine learning for code analysis and optimization in transpilers, several integration challenges remain:

- Balancing analysis depth with compilation speed
- Ensuring deterministic behavior in the presence of probabilistic ML models
- Managing the computational resources required for ML-based analysis
- Validating the correctness of ML-suggested optimizations

Addressing these challenges requires thoughtful architecture design and validation strategies that combine of machine learning with the reliability guarantees expected from compiler technology (Leather and Cummins, 2020).

3.3.4 Emerging Approaches

Recent research has explored several innovative approaches to incorporating machine learning into code analysis and optimization:

- Transfer learning from natural language to code understanding

- Unsupervised learning on vast code repositories to discover optimization patterns
- Interactive learning systems that incorporate developer feedback into optimization decisions
- Explainable AI techniques that help developers understand and trust ML-based optimizations

These emerging approaches hold significant promise for next-generation transpilers that can adapt to new languages and optimization contexts with minimal human intervention (Sharma et al., 2021).

3.4 Automated Refactoring

Automated refactoring is a powerful technique in software development that uses algorithms and tools to restructure existing code without changing its external behavior. By integrating machine learning (ML), this process becomes smarter and more efficient, particularly in transpilation—where code is translated from one programming language to another (Baqais and Alshayeb, 2020). Let's break this down into its definition, importance, and the ML approaches that make it work.

Definition

Automated refactoring involves the use of automated tools to identify and apply changes, known as refactorings, to improve a codebase's structure, readability, maintainability, or performance, all while keeping its functionality intact. Unlike manual refactoring, which depends on a developer's expertise and intuition, automated refactoring relies on systematic analysis and predefined patterns to suggest or implement these improvements (Baqais and Alshayeb, 2020).

Importance in Transpilation

When transpiling code from one language to another, simply ensuring functional equivalence isn't enough. Automated refactoring steps in to elevate the translated code to a higher standard. Here's why it's critical:

- **Enhanced Code Quality:** It ensures the translated code follows the target language's best practices and style conventions, making it more readable and easier to maintain. For example, it might reorganize messy, auto-generated code into a cleaner, more idiomatic form.
- **Performance Optimization:** Automated tools can apply optimizations like loop unrolling or function inlining, tailoring the code for better speed or resource efficiency in the new language.

- **Error Reduction:** By applying proven refactoring patterns consistently, automation reduces the chance of human error—crucial in large-scale migrations where manual adjustments could introduce bugs.

In essence, automated refactoring transforms transpilation from a basic translation task into an opportunity to deliver high-quality, optimized code, saving time and effort in big projects (Baqais and Alshayeb, 2020).

3.4.1 Machine Learning Approaches

Machine learning supercharges automated refactoring by enabling systems to learn from codebases, spot opportunities for improvement, and execute complex transformations. Here are the key ML techniques driving this process:

Pattern Recognition for Refactoring Opportunities

- **How It Works:** Supervised ML models are trained on datasets of code snippets labeled with "code smells"which means signs of suboptimal design, like long methods or duplicate code. These models then detect similar issues in new code and suggest refactorings, such as splitting a lengthy function into smaller ones.
- **Example:** A model might flag repeated code blocks and recommend consolidating them into a reusable function.

(Wang et al., 2013)

Sequence-to-Sequence Models for Code Transformation

- **How It Works:** Borrowing from natural language processing, transformer-based models learn from pairs of pre and post-refactored code. They can then generate refactored versions directly, improving structure or efficiency while preserving behavior.
- **Example:** Converting a chain of **if-else** statements into a **switch** statement or rewriting imperative loops as functional expressions in languages like *Python*.

(Peters et al., 2019)

Reinforcement Learning for Adaptive Refactoring

- **How It Works:** Reinforcement Learning (RL) trains agents to experiment with refactoring actions, using feedback from metrics like code complexity or test coverage to refine their approach. Over time, the agent learns the best strategies for complex, multi-step refactorings.
- **Example:** An RL agent might iteratively simplify a convoluted function, balancing readability and performance based on real-time feedback.

(Palit and Sharma, 2024)

Clustering for Code Style Harmonization

- **How It Works:** Unsupervised learning, such as clustering, groups similar code segments to spot inconsistencies or deviations from norms. In transpilation, this ensures the translated code matches the target project's style.
- **Example:** Identifying and standardizing variable naming conventions across a codebase for uniformity.

(Kuhn et al., 2007)

Together, these ML techniques make automated refactoring tools more precise, adaptable, and capable of handling the nuances of code translation.

3.5 Error Detection and Correction

The integration of machine learning techniques into transpilers has significantly enhanced their ability to detect and correct errors during the translation process. This section explores how artificial intelligence approaches are revolutionizing both static and dynamic error handling in transpilation systems.

3.5.1 Static Code Analysis with AI

Static code analysis examines code without execution to identify potential errors, bugs, and vulnerabilities. Traditional static analyzers rely on predefined rules and pattern matching, which often generate numerous false positives and may miss complex, context-dependent errors. Machine learning approaches overcome these limitations through more sophisticated understanding of code semantics and patterns (Louridas, 2006).

Deep Learning for Bug Detection

Neural network architectures, particularly those based on transformers and graph neural networks, have demonstrated remarkable capabilities in identifying bugs and potential vulnerabilities in source code. These models learn from vast repositories of code, including both correct implementations and those containing known bugs, allowing them to recognize patterns that traditional rule-based systems cannot detect (Allamanis, 2022).

- Representing code as Abstract Syntax Tree (AST) or control flow graphs
- Encoding code using embeddings that capture semantic relationships
- Applying attention mechanisms to focus on error-prone sections
- Predicting probability distributions over potential error types

These approaches have proven particularly effective for detecting language-specific errors that might occur during transpilation, such as type inconsistencies, memory management issues, and concurrency problems that differ between source and target languages (Xiao et al., 2023).

Probabilistic Error Localization

Beyond simply flagging potential errors, AI-enhanced static analysis can provide probabilistic assessments of where errors are most likely to occur in transpiled code. This capability allows developers to focus their verification efforts on high-risk sections of code and helps transpiler systems prioritize their error correction strategies (Shooman, 1972).

Bayesian models and uncertainty quantification techniques enable transpilers to express confidence levels in their translations, identifying sections of code where the translation is potentially ambiguous or error-prone. This information can be used to generate appropriate warnings or to trigger more intensive analysis for those sections (Turner et al., 2021).

Automated Repair Suggestion

The most advanced AI-based static analyzers go beyond detection to suggest potential repairs for identified issues. These systems leverage generative models trained on pairs of buggy and corrected code to propose fixes that:

- Preserve the original code's intended functionality
- Adhere to target language best practices and idioms
- Maintain consistency with the surrounding codebase
- Address the root cause rather than just symptoms of the error

For transpilers, this capability is particularly valuable when dealing with language-specific constructs that have no direct equivalent in the target language, requiring creative reformulation to maintain correctness (Monperrus, 2019).

3.5.2 Dynamic Analysis: Runtime Error Detection and Correction

While static analysis occurs before program execution, dynamic analysis examines program behavior during runtime, capturing errors that may only manifest under specific execution conditions or input patterns.

Learning from Execution Traces

Machine learning models can be trained on execution traces collected during program runs, learning to recognize patterns that precede failures or unexpected behaviors. In the context of transpilation, these models can compare execution traces between source and transpiled programs to identify semantic discrepancies that may indicate translation errors (Cornelissen et al., 2008).

Sequence models such as Long-Short-Term Memory (LSTM) and transformers excel at processing execution traces, identifying temporal patterns that might indicate:

- Memory access violations
- Concurrency issues
- Performance anomalies
- Unexpected control flow paths

By analyzing these patterns, machine learning systems can provide early warnings about potential runtime errors in transpiled code and suggest corrective measures.

Reinforcement Learning for Error Correction

Reinforcement learning approaches offer a powerful framework for dynamic error correction in transpilers. These systems can:

- Explore different correction strategies through trial and error
- Receive feedback based on program correctness and performance
- Learn optimal correction policies for different types of errors
- Adapt to new error patterns without explicit reprogramming

When integrated into transpilation workflows, reinforcement learning agents can continuously monitor program execution, intervene when errors are detected, and learn from the outcomes of their interventions to improve future error correction strategies (Gupta et al., 2019).

Self-Healing Code Generation

The most advanced dynamic analysis systems incorporate self-healing capabilities, automatically modifying transpiled code in response to runtime errors. These systems leverage online learning techniques to:

- Detect failures during program execution
- Diagnose the root causes of these failures
- Generate patches that address the underlying issues
- Verify the effectiveness of these patches through continued monitoring

For transpilers, self-healing capabilities are particularly valuable in production environments where manual intervention may not be feasible, allowing automatically translated code to adapt to unforeseen runtime conditions (Monperrus, 2014).

3.5.3 Integration of Static and Dynamic Approaches

The most effective error detection and correction systems in modern transpilers combine static and dynamic analysis, leveraging the complementary strengths of both approaches.

Continuous Learning Pipelines

Machine learning enables the creation of continuous learning pipelines that integrate information from both static and dynamic analysis:

- Static analysis provides initial error predictions and correction suggestions
- Dynamic analysis validates these predictions and collects data on missed errors
- Feedback from both sources is used to retrain and improve the underlying models
- The updated models inform future static analysis, creating a virtuous cycle

This integrated approach allows transpilers to continuously improve their error detection and correction capabilities through real-world usage.

Multi-Modal Error Representation

Advanced AI systems for error detection and correction leverage multi-modal representations that combine:

- Syntactic information from code structure
- Semantic information from type systems and data flow
- Temporal information from execution traces
- Contextual information from surrounding code and documentation

These rich representations enable more accurate error detection and more effective correction strategies by considering the full context in which errors occur (Chakraborty and Ray, 2021).

3.5.4 Challenges and Limitations

Despite significant advances, several challenges remain in applying machine learning to error detection and correction in transpilers:

- Balancing precision and recall in error detection
- Ensuring that corrections preserve intended program semantics

- Managing the computational overhead of complex analysis
- Handling rare or previously unseen error patterns
- Providing explanations for detected errors and suggested corrections

Addressing these challenges represents an active area of research at the intersection of machine learning, program analysis, and transpiler technology.

3.5.5 Future Directions

Emerging research in error detection and correction for transpilers points to several promising directions:

- Few-shot learning approaches that can quickly adapt to new languages and error types
- Explainable AI techniques that help developers understand detected errors and proposed corrections
- Collaborative systems that effectively combine machine learning with human expertise
- Domain-specific models tailored to particular application areas or programming paradigms

These advances promise to further enhance the reliability and effectiveness of machine learning-enhanced transpilers, making cross-language code translation increasingly robust and trustworthy.

3.6 Cross-Linguistic Translation

The process of translating code between different programming languages presents unique challenges that traditional rule-based transpilers often struggle to address effectively. This section examines how machine learning approaches offer novel solutions to these challenges, enabling more robust and adaptable cross-linguistic translation (Xiao and Guo, 2013).

3.6.1 AI Solutions for Syntactic Divergence

Programming languages exhibit significant syntactic differences that complicate direct translation. Machine learning models address these challenges through sophisticated pattern recognition and transformation capabilities.

Neural Sequence-to-Sequence Translation

Neural sequence-to-sequence models have revolutionized code translation by treating source code as a specialized language with its own grammar and semantics. These models, particularly those based on transformer architectures, excel at:

- Learning alignments between syntactic constructs across languages
- Generating idiomatic translations that conform to target language conventions
- Handling variable-length input and output sequences
- Preserving the semantic intent of code despite syntactic differences

Pre-trained language models fine-tuned on parallel code corpora have demonstrated remarkable ability to translate between languages with divergent syntax, such as translating between procedural and functional programming paradigms. These models effectively learn to map equivalent constructs even when they bear little surface similarity (Raina et al., 2024) Peters et al. (2019).

Abstract Syntax Tree Transformations

Machine learning approaches that operate on ASTs provide a powerful mechanism for handling syntactic differences. Deep learning models can:

- Encode source code ASTs into latent representations that capture structural information
- Transform these representations to align with target language constraints
- Decode the transformed representations into well-formed target language ASTs
- Preserve the hierarchical relationships between code elements during translation

Graph neural networks are particularly effective for this task, as they naturally capture the tree structure of ASTs while enabling complex transformations that maintain code validity and structure (Jiang et al., 2021).

3.6.2 AI Approaches to Semantic Preservation

Beyond syntax, transpilers must preserve program semantics across languages with different execution models, type systems, and standard libraries.

Type-Aware Translation Models

Machine learning models enhanced with type information can ensure semantic correctness when translating between languages with different type systems. These models:

- Infer types in weakly-typed source languages
- Generate appropriate type annotations in strongly-typed target languages
- Handle implicit type conversions and prevent type-related errors
- Maintain type safety across language boundaries

Probabilistic type inference systems powered by machine learning can resolve ambiguities in dynamically-typed languages, enabling more accurate translation to statically-typed languages (Hellendoorn et al., 2018).

Runtime Behavior Modeling

To address differences in execution semantics, AI approaches incorporate models of runtime behavior:

- Execution trace analysis identifies behavioral patterns in source code
- Generative models translate these patterns into equivalent constructs in the target language
- Reinforcement learning optimizes for behavioral equivalence under various execution conditions
- Verification techniques confirm semantic preservation through symbolic execution or test generation

These techniques are particularly valuable when translating between languages with different memory models, concurrency mechanisms, or exception handling approaches (Hellendoorn et al., 2018).

3.6.3 AI Solutions for Library and Ecosystem Adaptation

Different programming languages exist within distinct ecosystems with their own standard libraries, frameworks, and conventions.

API Mapping Through Representation Learning

Machine learning models can learn mappings between equivalent Application Programming Interface (APIs) across different language ecosystems:

- Embedding models capture functional similarities between library functions
- Attention mechanisms identify contextual usage patterns
- Few-shot learning adapts to new library versions or frameworks
- Multi-modal models incorporate documentation and examples to improve mapping accuracy

These approaches enable transpilers to automatically suggest appropriate library substitutions when direct equivalents are unavailable, significantly reducing the manual effort required for ecosystem adaptation (Lyu et al., 2021).

Contextual Code Generation

AI-powered transpilers leverage contextual understanding to generate appropriate code in the target ecosystem:

- Language models condition code generation on both source code and target ecosystem conventions
- Transfer learning from large code corpora captures idiomatic patterns in the target language
- Retrieval-augmented generation incorporates relevant examples from the target ecosystem
- Adaptation techniques handle evolving libraries and frameworks

These capabilities allow transpilers to produce not just syntactically correct but also idiomatic and maintainable code that aligns with target language best practices (Wang et al., 2021).

3.6.4 AI-Enabled Handling of Language-Specific Features

Programming languages often contain unique features that have no direct equivalents in other languages, such as *Python*'s list comprehensions, *Rust*'s ownership system, or *C# LINQ* expressions.

Feature Decomposition and Reconstruction

Machine learning approaches decompose complex language-specific features into their semantic components and reconstruct equivalent functionality:

- Semantic parsing identifies the underlying computational intent
- Decomposition models break features into fundamental operations
- Reconstruction algorithms assemble equivalent implementations in the target language
- Optimization ensures performance characteristics are preserved

This approach enables translation of idiomatic code that leverages language-specific features without requiring manual reimplementation (Lee et al., 2021).

Analogical Reasoning for Feature Mapping

Neural models can perform analogical reasoning to identify equivalent patterns across languages:

- Embedding spaces capture functional similarities despite syntactic differences
- Contrastive learning identifies analogous constructs across language boundaries
- Few-shot exemplar models learn from minimal examples of equivalent implementations
- Multi-headed attention mechanisms capture multiple valid translation alternatives

These techniques allow transpilers to handle novel language features by drawing analogies to known patterns, even in the absence of explicit translation rules (Hoc and Nguyen-Xuan, 1990).

3.7 Case Studies and Applications

The integration of machine learning techniques into transpilers has led to significant advancements in cross-language code translation. This section examines real-world applications and case studies that demonstrate the practical impact of AI-enhanced transpilers across various domains and programming paradigms.

3.7.1 Real-World Examples

Neural Machine Translation for JavaScript Modernization

Several projects have applied neural machine translation techniques to modernize legacy JavaScript code. These systems leverage transformer-based architectures to convert *ES5 JavaScript* to modern *ES6+* syntax automatically:

- Transformer models fine-tuned on *JavaScript* code repositories successfully identify and replace outdated constructs with modern equivalents
- Attention mechanisms focus on context-dependent transformations that require understanding of code semantics
- Pre-trained language models adapted for code translation maintain code correctness while improving maintainability
- Evaluation metrics show significant improvements in readability and performance compared to manually updated code

These applications demonstrate how machine learning approaches can automate tedious modernization tasks while preserving application functionality and enhancing code quality (Paltoglou et al., 2021).

Cross-Language Migration Systems

Several industrial and research projects have developed machine learning-powered systems for migrating entire codebases between programming languages:

- **Python-to-Java** transpilers using deep learning to handle *Python*'s dynamic typing in *Java*'s statically typed environment
- **JavaScript-to-TypeScript** conversion systems that leverage probabilistic type inference to add appropriate type annotations
- **C/C++-to-Rust** translators that apply reinforcement learning to address memory safety concerns
- **COBOL-to-Java** migration tools that combine neural networks with domain-specific knowledge to modernize legacy enterprise applications

These systems have demonstrated cost savings of up to 70% compared to manual rewrites while maintaining comparable code quality and performance characteristics (Gui et al., 2022).

Domain-Specific Language Translation

Machine learning has proven particularly effective for translating Domain-Specific Languages (DSL) to general-purpose languages:

- *SQL*-to-code generators that translate database queries into equivalent application code
- Shader language transpilers that convert between OpenGL Shading Language (GLSL), High-Level Shader Language (HLSL), and platform-specific variants
- Hardware description language translators that enable cross-platform hardware development
- Mathematical modeling language converters that transform domain-specific notations into executable code

These specialized transpilers leverage both the structure of the domain-specific language and machine learning to produce highly optimized translations that would be difficult to achieve with rule-based approaches alone (Bhatia et al., 2023).

3.7.2 Success Stories and Challenges

Enterprise Legacy System Modernization

Several large organizations have successfully applied AI-enhanced transpilers to modernize critical legacy systems:

- Financial institutions have used ML-powered transpilers to migrate millions of lines of *COBOL* code to *Java* or *C#*
- Healthcare systems have employed neural translation to update legacy medical record processing software
- Government agencies have leveraged machine learning to modernize administrative systems while preserving decades of business logic
- Telecommunications companies have applied *AI*-based code translation to migrate embedded systems code to modern platforms

These projects have reported significant benefits, including reduced maintenance costs, improved performance, better security, and easier recruitment of developers familiar with modern languages.

Open Source Translation Frameworks

The open-source community has embraced machine learning approaches for transpilation, resulting in several notable projects:

- Frameworks that combine rule-based approaches with neural networks to achieve higher accuracy
- Community-maintained models trained on diverse codebases to handle a wide range of coding styles
- Plugin ecosystems that extend translation capabilities to specialized libraries and frameworks
- Interactive tools that allow developers to guide the translation process with minimal intervention

These projects have democratized access to advanced code translation technologies, enabling smaller organizations to benefit from the same approaches used by larger enterprises.

Performance and Correctness Trade-offs

The analysis of case studies reveals important trade-offs between performance and correctness in the application of machine learning to transpilation. Neural network inference can introduce latency in the process, affecting the speed of code conversion. Additionally, since probabilistic models may produce uncertain results, verification strategies are necessary to ensure translation accuracy. The quality of the translation can also vary depending on the similarity between the source and target languages, impacting the precision of the conversion. Another critical factor is domain-specific knowledge, which remains essential for handling specialized code patterns. Successful implementations have addressed these challenges through hybrid approaches that combine machine learning with traditional compilation techniques, leveraging the strengths of both methods.

3.7.3 Industry-Specific Applications

Financial Technology

The financial sector has been an early adopter of ML-enhanced transpilers, focusing on:

- Risk modeling code translation between specialized mathematical languages and production environments
- High-frequency trading system migration between different platforms and languages
- Regulatory compliance through automated code updates to meet changing requirements
- Legacy banking system modernization while preserving critical business logic

These applications have demonstrated particular value in maintaining the correctness of financial calculations while improving maintainability and performance (Kripets, 2024).

Scientific Computing

Machine learning approaches have revolutionized scientific code translation:

- **FORTRAN-to-Python/Julia** transpilers that modernize legacy scientific code
- Cross-platform scientific application deployment through targeted code generation
- Mathematical notation to code translation for accelerated research implementation

These tools have significantly reduced the barriers between theoretical research and practical implementation, accelerating scientific progress across disciplines (Heath, 2018).

Mobile and Web Development

The mobile and web development ecosystem has benefited significantly from AI-powered transpilation. Cross-platform frameworks leverage neural code translation to achieve native performance, enabling seamless execution across different operating systems. Progressive web application converters facilitate the transformation of web-based code into mobile-optimized implementations, ensuring efficient performance on handheld devices.

Additionally, responsive design generators dynamically adapt code to accommodate various device capabilities, enhancing usability across screens of different sizes. AI-driven transpilation has also contributed to accessibility improvements by automating code transformations that enhance compatibility with assistive technologies.

These advancements have collectively reduced development time and costs while significantly improving the user experience across diverse platforms and devices (Andrés and Pérez, 2017).

3.7.4 Quantitative Impact Assessment

Productivity Metrics

Studies measuring the impact of ML-enhanced transpilers on development productivity have reported:

- 40-60% reduction in migration project timelines compared to manual rewrites
- 20-40% decrease in post-migration defects when using AI-assisted translation
- 10-25% improvement in developer productivity during code maintenance
- Significant reduction in onboarding time for developers working with translated codebases

These productivity gains translate directly to cost savings and faster time-to-market for organizations undertaking language migration projects (Peng et al., 2023) (Machinet, 2023) (Tabachnyk and Nikolov, 2022).

Quality Assessments

Evaluations of code quality metrics show mixed but generally positive results:

- Readability scores of machine-translated code approach 80-90% of human-written equivalents
- Performance characteristics vary, with some ML-transpiled systems achieving numbers really close to the original performance
- Maintainability indices typically improve by 15-25% compared to legacy code
- Security vulnerability shows reduction through translation to safer language constructs but there have been some studies that identified that AI-assisted code translation can introduce bugs, some of which may have security implications.

These metrics indicate that machine learning approaches can produce high-quality translations that meet or exceed manual conversion efforts in many dimensions (Alawad et al., 2019).

3.8 Future Directions

The integration of artificial intelligence into transpilation is set to advance significantly with emerging machine learning technologies. Deep learning, particularly through transformer models, offers substantial potential due to its proven ability to model complex relationships in data, such as those found in programming languages. These models could improve the precision and speed of code translations across diverse languages, adapting to intricate syntax and semantic patterns. Similarly, Large Language Models (LLMs) like GPT-4 or CodeBERT, when tailored to transpilation, could deliver highly context-sensitive translations that align with specific project requirements or developer preferences.

Innovative learning approaches also promise to reshape transpilation. Reinforcement Learning (RL), for instance, could enable transpilers to dynamically refine their translation strategies by learning from feedback on code quality metrics, such as execution efficiency or maintainability. Additionally, unsupervised and semi-supervised learning techniques could tap into vast, unlabeled code repositories, overcoming limitations where paired translation datasets are scarce.

Beyond technical progress, ethical considerations must remain central. Issues like safeguarding data privacy, minimizing algorithmic bias, and ensuring the security of AI-generated code will be critical to responsibly advancing these tools. Progress in this field will hinge on collaboration across machine learning, software engineering, and programming language research to tackle both opportunities and challenges effectively.

3.9 Summary

Artificial Intelligence (AI) significantly enhances transpilers by overcoming the limitations of traditional rule-based systems. Through techniques like machine learning and natural language processing, AI enables transpilers to better understand code, perform more accurate translations, and handle complex language features efficiently.

These advancements lead to improved code mapping, context-aware translations, and automated refactoring capabilities. Real-world applications, such as tools like GitHub Copilot or automated refactoring systems, demonstrate AI's tangible impact, making transpilation faster, more reliable, and broadly applicable in software development.

The integration of AI in transpilation opens up exciting avenues for further exploration. Future research could focus on developing advanced models capable of translating a wider range of programming languages or adapting to specific project requirements. Emerging areas, such as using reinforcement learning to dynamically optimize translation strategies or addressing ethical concerns like bias and security in AI-generated code, hold significant potential for improvement.

For developers, AI-enhanced transpilers must be practical—transparent, trustworthy, and easy to integrate into existing workflows—to ensure widespread adoption and effectiveness. Continued innovation in this field promises to redefine transpilation and elevate software engineering as a whole.

Chapter 4

System Proposal

This chapter proposes the architecture of a system designed to facilitate a systematic comparative analysis between machine learning-based approaches and traditional transpilation methods for code translation. The proposed framework integrates multiple components intended to enable evaluation of both translation methodologies, focusing on performance metrics, accuracy assessment, and resource utilization. Through careful system design, it is established a controlled environment for examining the relative strengths and limitations of each approach. The architecture described here represents a novel integration of traditional software engineering principles with contemporary machine learning techniques.

In the following sections, it is provided a detailed description of each architectural component, beginning with an overview of the system's structure and data flow. The discussion then progresses through individual components, exploring their specific roles, interactions, and contributions to the overall evaluation framework. This analysis establishes the foundation for the experimental methodology and subsequent result analysis.

Due to the exploratory and still abstract nature of this project, no requirements have yet been defined. This includes functional and non functional ones as well. It means that a traditional system proposal that includes the creation of other Unified Modeling Language (UML) diagrams (Use Case, Entity Relationship, Activity, etc.) is not possible.

4.1 Software Architecture Diagram

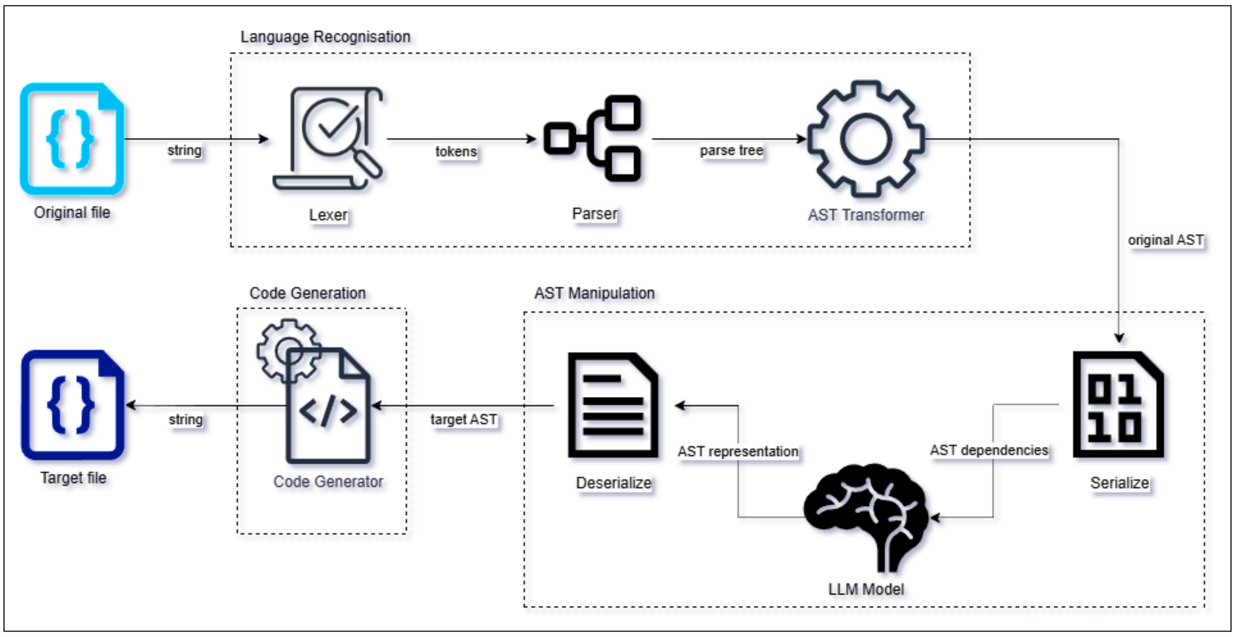


Figure 1: System Architecture

The figure 1 illustrates the overall architecture of the proposed system, which integrates traditional transpilation techniques with a generative Large Language Model to enhance the code translation process. The workflow is divided into three main stages — Language Recognition, AST Manipulation, and Code Generation — each responsible for a distinct part of the transformation pipeline. The process begins with the Original File, which is read as a string and passed through a series of components that analyse and transform its structure before producing the Target File in the desired programming language.

4.1.1 Language Recognition

The Language Recognition module implements the first processing stage of the transpilation system. This component consists of three primary elements that work in sequence to transform the source code into a structured abstract representation.

The Lexer serves as the first point of contact with the input file, processing the raw source code string into a sequence of tokens. These tokens represent the smallest meaningful units of the programming language, such as keywords, identifiers, operators, and literals. This tokenization process effectively breaks down the source code into manageable, atomic components.

Following tokenization, the Parser analyzes the sequence of tokens according to the formal grammar rules of the source programming language. Processes these tokens to construct a parse tree that represents the hierarchical syntactic structure of the program. This step ensures that the code complies with the language's syntactic rules and prepares it for further processing.

The AST Transformer then converts the parse tree into an AST, a higher-level representation that captures the essential structure and semantics of the program while abstracting away unnecessary syntactic details. This transformation produces the original AST, which serves as input for the subsequent manipulation phase.

4.1.2 AST Manipulation

The AST Manipulation module implements the core transformation process where the original AST is converted into the target language representation. This phase takes advantage of both traditional programming techniques and machine learning capabilities to achieve accurate code translation.

The Serialize component takes the original AST and converts it into a format suitable for processing by the LLMs. This serialization ensures that the structural and semantic information contained in the AST can be processed effectively by the neural network.

The serialization process in the AST Manipulation phase is crucial for enabling effective communication between the Abstract Syntax Tree and the Large Language Model. By transforming the hierarchical AST structure into a linear, model-friendly format, we preserve the intricate syntactic and semantic relationships of the original code while making it compatible with neural network architectures.

At the heart of this phase lies the LLMs, which analyzes the AST dependencies and patterns to understand the program's structure and intent. The model in question has not been defined for now but there already some options in the table like: GPT-4, Claude 3.5 Sonnet, LLaMA 3.3 Code and the new DeepSeek-R1.

The Deserialize component then processes the LLMs output, reconstructing a target AST that represents the program in the desired target language while preserving the original program's functionality and structure.

4.1.3 Code Generation

The last module of the proposed architecture transforms the processed AST back into human-readable source code in the target programming language. The Code Generator component takes the target AST as input and applies language-specific formatting rules and syntax to produce the final source code file.

This component ensures that the generated code not only maintains functional equivalence with the original program, but also adheres to the conventions and best practices of the target language. The output is written to a target file, completing the transpilation process.

4.2 Input and Output Specifications

The transpilation system operates on complete source code files rather than isolated code snippets, representing a more comprehensive and practical approach to code translation. This design choice reflects real-world software development scenarios where entire programs or modules require translation between programming languages.

The input to the system consists of well-formed source code files containing complete program implementations. These files may include multiple classes, functions, and dependencies, allowing the system to handle complex program structures and maintain contextual relationships throughout the translation process. By processing entire files, the system can better understand and preserve program architectures, import relationships, and scope hierarchies that might be lost when translating code fragments in isolation.

On the output side, the system generates complete, executable source files in the target programming language. These output files maintain the structural integrity of the original program while adhering to the conventions and best practices of the target language. The generated files include necessary language-specific elements such as import statements, package declarations, and proper file extensions, ensuring they can be immediately integrated into existing development workflows.

This file-based approach offers several advantages over snippet-based translation:

- Preservation of program context and scope;
- Maintenance of inter-component relationships;

- Complete handling of dependencies;
- Generation of production-ready code artifacts;
- Seamless integration with existing development tools and processes.

The system's ability to process complete source files makes it particularly valuable for large-scale software modernization projects where entire codebases need to be migrated between programming languages while maintaining their functional integrity and structural organization.

4.3 System Process Flow

The transpilation process follows a systematic workflow, depicted in Figure 2 that transforms the source code through multiple stages of analysis and transformation. Understanding this process flow is crucial for understanding how the system maintains code integrity throughout the translation process.

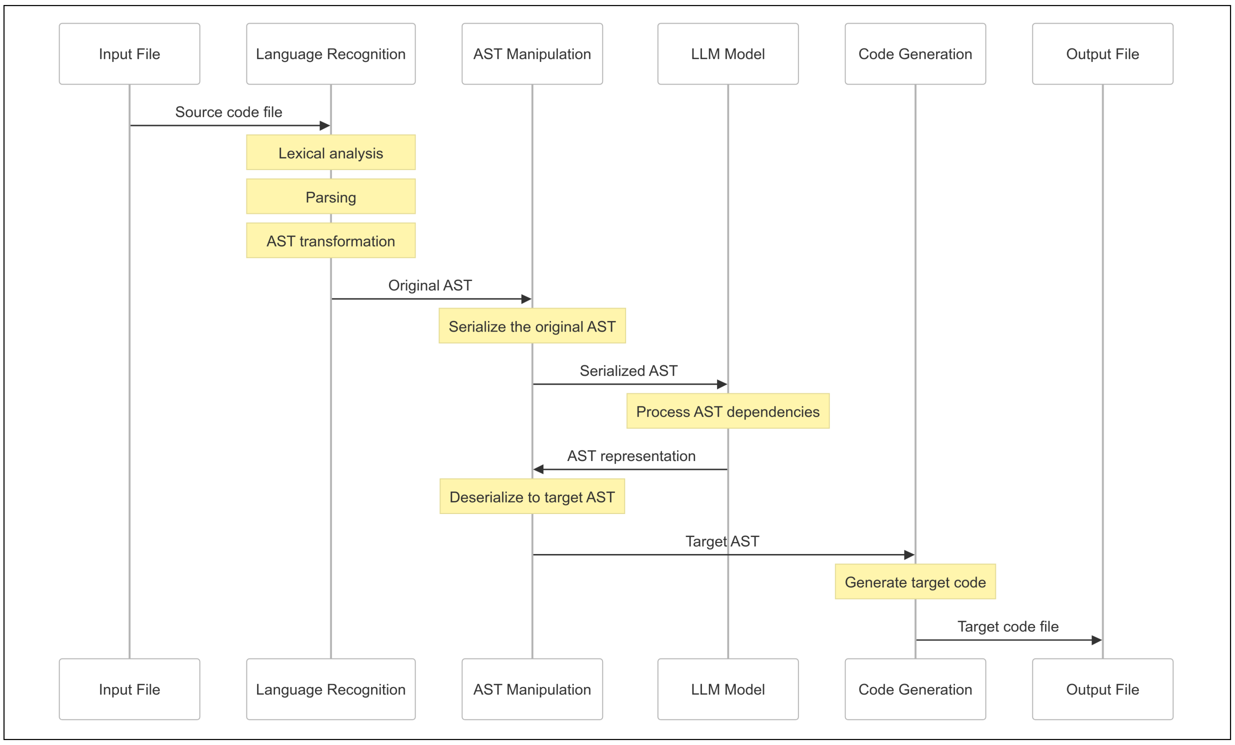


Figure 2: Representation of the system's flow

The process begins when a source code file enters the Language Recognition phase. During this initial stage, the system performs lexical analysis to break down the source code into tokens, followed by parsing these tokens into a structured format. The parser then works in conjunction with the AST Transformer to create an abstract syntax tree that represents the program's structure.

Once the original AST is generated, the system transitions to the AST Manipulation phase. This critical stage involves preparing the AST for processing by the Large Language Model. The AST undergoes serialization that converts it into a format suitable for the LLMs input requirements. The LLMs processes this input, analyzing the program's structure and generating an appropriate representation for the target language.

The manipulation phase continues as the system deserializes the LLMs output, constructing a new AST that conforms to the target language's specifications. This transformation preserves the program's semantic meaning while adapting its structure to align with the target language's paradigms and conventions.

In the final stage, the Code Generation phase takes the transformed AST and produces human-readable source code in the target language. This generated code maintains the functionality of the original program while adhering to the target language's syntax and best practices. The process ends the system writing the translated code to an output file, completing the transpilation cycle.

This systematic approach to code translation, combined with the integration of machine learning capabilities, provides a framework to handle complex code transformations while maintaining the functionality and structure of the program.

Chapter 5

Transpilers - An independent research

The research performed on various transpilers provides valuable information on the performance, accuracy and usability of these tools. Analyzing the characteristics and results of the tests executed allows several relevant conclusions to be identified, highlighting their advantages, limitations, and ideal contexts of use. Next are presented the main observations drawn from the research, focusing on efficiency, accuracy, resource consumption, and practical applicability.

5.1 Background to the Tests Performed

To test each transpiler were used code snippets that implement the same functional logic, processing a list of numbers to determine the sum of even numbers and find the largest number, but are tailored to the input languages that each tool supports, including **JavaScript/TypeScript, Nim, Clojure, Haxe, C, and Java** [1]. A fair and consistent evaluation was made possible by the fact that despite the syntactic and structural variations among the snippets, the underlying logic was always the same. The consistency of the test logic allowed for a comparative analysis of each transpiler's performance, accuracy, and efficiency and the results focused on the unique features of the tools rather than differences in the input code's complexity or intent.

[1] https://justandre02.github.io/Comparative-Review-and-Empirical-Evaluation-about-transpilers/

5.2 Transpiler Selection and Challenges

Tool	Input Languages	Output Languages	AST Handling	Ease of Use	Stability & Known Issues
TypeScript Compiler	TypeScript	JavaScript	Complete	Simple	High stability, few errors
Babel	JavaScript, TypeScript	JavaScript	Complete	Intuitive	High stability, occasional errors
eslint	JavaScript, TypeScript	–	Partial	Moderate	High stability, frequent reports
Nim	Nim	C, C++, JavaScript	Complete	Moderate	Stable, actively developed
ClojureScript	Clojure	JavaScript	Complete	Moderate	Stable, some reported errors
jscodeshift	JavaScript, TypeScript	JavaScript	Complete	Easy	Moderately stable, occasional errors
ast-grep	JavaScript, TypeScript	–	Simple	Moderate	Moderately stable
Haxe	Haxe	JavaScript, Python	Complete	Moderate	Stable, frequent updates
c2rust	C	Rust	Complete	Moderate	Stable, but complex bugs
Fennel	Fennel (Lisp dialect)	Lua	Complete	Simple	Stable, sporadic maintenance
Cito	C	JavaScript	Partial	Moderate	Moderately stable
TypeScriptToLua	TypeScript	Lua	Complete	Moderate	Stable, minor known bugs
godzilla	JavaScript	Lua	Complete	Moderate	Moderately stable
jsweet	Java	JavaScript, TypeScript	Complete	Moderate	Moderately stable, some bugs
j2cl	Java	JavaScript	Complete	Moderate	Stable, interoperability bugs
Java2Python	Java	Python (2.x)	Simple	Easy	Stable, but obsolete (Python 2 only)

Table 1: Key characteristics of the selected transpilers.

The transpilers analyzed in this investigation were selected based on their popularity on *GitHub*, assessed by the number of stars they received, which reflects the adoption and interest of the developer community. The transpilers that stood out most in this criterion were prioritized, guaranteeing the relevance and technical support of the tools chosen. In addition, **Java2Python** was incorporated into the list, although it was not among the most popular, as it was the only option identified capable of converting **Java** code to **Python**, meeting a specific need of the study.

However, running the tests faced significant limitations, since not all selected transpilers could be installed and run on the *Linux* system used. These difficulties were mainly caused by outdated dependencies or incompatibilities with the *Linux* environment, restricting the experiments to a subset of the tools originally planned.

5.2.1 Collumn Definitions

In this table, the **Tool** column lists the name of each transpiler under evaluation for reference. The **Category** column classifies each tool according to its conversion type—*"Language-to-Language"* for those translating between different languages, *"Version"* for those handling differences between versions of the same language, and *"Paradigm"* for those mapping between distinct programming paradigms. The **Input Languages and Output Languages** columns indicate, respectively, which source languages the tool accepts and which target languages it can generate. The **AST handling** column describes the extent of AST support: *"Complete"* denotes full coverage of nodes and complex transformation passes; *"Partial"* signifies limited support for specific nodes or operations; and *"Simple"* refers to basic functionality for pattern matching or data extraction without deep rewriting capabilities. The Ease of Use column provides a qualitative assessment of the learning curve and interface clarity: *"Simple"* or *"Easy"* indicates straightforward, well-documented APIs requiring minimal configuration; *"Intuitive"* implies a coherent workflow that is naturally accessible despite some complexity; and *"Moderate"* signals that additional configuration steps or intermediate concepts must be mastered before the tool can be utilised to its full potential and *"Hard"* signifies that independent research must be conducted before using the tool. And finally the **Stability & Known Issues** column summarises the general maturity and reliability of each tool, noting whether it is actively maintained, prone to occasional errors or complex bugs, and any recurring issues that users should be aware of.

5.3 Results

Table 2 summarizes the four parameters, *execution time, CPU used, memory consumed, and accuracy*, measured to compare the eleven Transpilers chosen to conduct the experiment described in this chapter. The following subsections analyze performance, precision, resource consumption, and usability in detail. The last subsection summarizes the conclusions of the study. study.

Tool	Execution Time (s)	CPU Utilisation (%)	Memory Used (MB)	Accuracy (0-100)
TypeScript Compiler	1.68	0.97	180.2	91.19
Babel	3.17	0.26	89.5	91.19
Nim (-> C)	0.61	1.01	77.8	100
Nim (-> C++)	0.59	0.99	79.4	100
Nim (-> JS)	0.12	0.90	24.5	100
ClojureScript	27.44	0.62	36.0	57.70
jscodeshift	0.06	2.35	1.76	68.02
c2rust	0.16	0.81	135.3	62
Fennel	0.10	0.98	7.8	80
TypeScriptToLua	5.93	0.59	302.3	63.57
Java2Python	0.09	0.25	39.9	100

Table 2: Performance, resource usage and accuracy results for each transpiler.

5.3.1 Efficiency and Performance

The transpilers were evaluated in terms of performance according to three parameters: execution time, CPU usage and RAM used. The Nim tool, which transpiles from **Nim** to **C, C++** and **JS**, stands out for its high performance. With recorded execution times of 0.61 seconds, 0.59 seconds and 0.12 seconds, respectively, and moderate memory consumption (77.8Mb, 79.4Mb and 24.5Mb), **Nim** proves to be an exceptionally fast and lightweight solution. In contrast, **ClojureScript** has the longest execution time, 27.44 seconds, associated with a relatively low memory utilization of 36.0Mb, suggesting unsuitability for tasks with critical time constraints. On the other hand, **jscodeshift** offers remarkable speed (0.06 seconds) and minimal memory consumption (1.76Mb), positioning itself as a viable option for fast and light transformations, albeit with less precision.

Most of this metrics were retrieved using the *time* tool in **Linux**, this is a very light tool which means that its usage basically doesn't impact the performance of the hardware when doing this experiment. In some cases this tool was not possible to use so a script developed in **JS** or **Python** was made using libraries integrated in the respective languages that allow to access the components and register the metrics needed.

5.3.2 Output Precision

Precision is an essential criterion when choosing a transpiler, and a significant variation in results was observed. **Nim** and **Java2Python** achieve maximum precision scores (100/100) in their respective outputs, demonstrating the production of highly reliable code. **TypeScript Compiler** and **Babel**, widely adopted tools in the **JavaScript** ecosystem, achieve a solid score of 91.19/100, reflecting their maturity and robustness. However, tools such as **ClojureScript** (57.70/100), **c2rust** (62/100) and **TypeScriptToLua** (63.57/100) show lower levels of accuracy, indicating that their results may need further validation or correction, which could jeopardize their use in production environments.

These scores were obtained using scripts that evaluate the output code in categories such as syntax, semantic, and code structure. The scripts also compares the output code to the original one in order to understand if the logic in the output follows the one in the source code. All of these tasks combine into a score from 0 to 100.

5.3.3 Resource Usage

Resource consumption varies substantially between transpilers, influencing their suitability for different contexts. **TypeScriptToLua** has the highest memory consumption, with 302.3Mb, followed by **TypeScript Compiler** (180.2Mb) and **c2rust** (135.3Mb), which can be challenging on systems with limited resources. In contrast, **jscodeshift** (1.76Mb) and **Fennel** (7.8Mb) show minimal memory consumption, offering efficiency for smaller-scale projects. CPU usage remains below 100 per cent for most of the tools, with the exception of **jscodeshift** (235 per cent), suggesting that processing demands are generally not a limiting factor in these tests.

Why in some cases the CPU usage exceeded 100% ?

CPU usage can exceed 100 percent because modern systems have several CPU cores, and the percentage is calculated in relation to the capacity of a single core. When the percentage gets above 100 percent, it means that the process is using several CPU cores simultaneously.

In the case of **jscodeshift**, CPU usage reaching 235% means that the transformation effectively uses around 2.35 CPU cores on average. This is possible because:

- **Node.js** (which runs **jscodeshift**) is multi-threaded through its event loop and job threads. The **V8 JavaScript** engine (used by **Node.js**) can parallelize certain operations.

5.3.4 Usability and Stability

Ease of use and stability are key aspects for programmer adoption and long-term reliability. **Babel** and **jscodeshift** stand out for their simplicity of writing, supported by ecosystems of plugins that extend their flexibility. **TypeScript Compiler** and **Babel** are also highly stable, with a low incidence of errors, and are robust options for consistent performance. **Nim**, although stable, is under active development and may undergo future changes. On the other hand, **Java2Python**, despite its perfect precision and stability, is hampered by obsolescence, as it only supports **Python 2**, making it nonviable for contemporary applications.

5.4 Conclusion

In short, choosing the right transpiler always depends on balancing performance, precision and resource consumption with the specific requirements of each project. For workflows centered on **JavaScript/-TypeScript**, **Babel** and the **TypeScript** compiler stand out for their compromise between robustness and versatility, offering competitive response times and low memory overhead. When raw performance is critical, particularly in systems programming or contexts with high data throughput, the **Nim** compiler is the most suitable option, thanks to its speed and efficient use of *CPU* and *RAM*.

On the other hand, tools such as **jscodeshift** are ideal for one-off light transformations, where agility of execution overrides the need for perfection in the result. In specialized situations, for example, migrations from **C** to **Rust** or from **Fennel** to **Lua**, solutions such as **c2rust** and **Fennel** offer a valuable starting point, although it is advisable to plan a manual validation phase for the artifacts generated, given their intermediate precision.

Ultimately, no transpiler is universally **'the best'**; each one has a set of advantages and limitations. The final recommendation is to carefully analyze the application context, test tools in advance on a representative subset of the source code, and evaluate key metrics *(execution time, memory usage, and output fidelity)*. In this way, it is possible to ensure an optimized and safe transition between languages, minimizing risks, and maximizing productivity.

Chapter 6

Implementation of the AST Translation System

This chapter presents a comprehensive technical account of the design, architecture, and implementation of the automated translation system developed for this thesis. The primary objective of this work is to explore the automation of translation between **Java** source code to modern, idiomatic **Python**.

The approach of this system diverges significantly from traditional rule-based transpilers that operate on a purely lexical, semantic and sintatic level. Instead, the core of this methodology is the manipulation of code at a higher level of abstraction. The system leverages Abstract Syntax Tree (AST) to create a structured, semantically aware representation of the source Java code. This AST is then transformed into an equivalent **Python** AST through the advanced pattern recognition and generative capabilities of a large language model. By operating on the syntactic structure of the code rather than its raw text, the system can achieve a more accurate and contextually appropriate translation. The following sections will provide a detailed breakdown of the system's architecture, its multi-phase translation pipeline, and the specific strategies employed to handle the complexities of this transformation.

6.1 System Architecture and Core Technologies

The translator is designed as a modular, multi-stage pipeline. This architectural choice ensures a clear separation of concerns, where each component performs a distinct, well-defined transformation on the input data. The system processes code by passing it through a sequence of components, starting with raw source text and progressively enriching and transforming it until the final target code is generated. This linear flow, depicted in Figure 3, allows for easier testing, maintenance, and future extension of each individual stage.

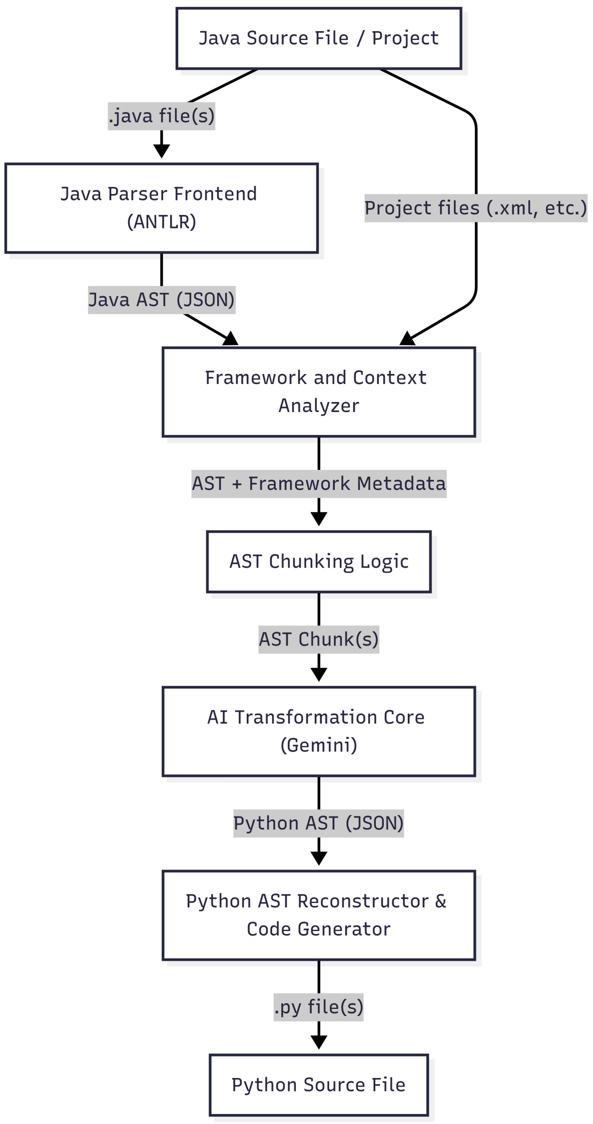

Java Source File / Project
.java file(s)
Java Parser Frontend (ANTLR)
Project files (.xml, etc.)
Java AST (JSON)
Framework and Context Analyzer
AST + Framework Metadata
AST Chunking Logic
AST Chunk(s)
AI Transformation Core (Gemini)
Python AST (JSON)
Python AST Reconstructor & Code Generator
.py file(s)
Python Source File

Figure 3: Data flow diagram of the translation system architecture

6.1.1 Core Components

The architecture is implemented through four primary software components, each responsible for a critical phase of the translation process.

The Java Parser Frontend

The entry point to the pipeline is the *Java Parser Frontend*. The primary responsibility of this component is to convert the unstructured text of a Java source file into a structured, machine-readable format. This is accomplished by invoking a parser, generated by the ANTLR tool from a formal Java grammar. The parser performs a lexical and syntactic analysis of the source code, producing a detailed parse tree. This tree is then traversed to construct a more abstract representation, the Abstract Syntax Tree, which is subsequently serialized into a JavaScript Object Notation (JSON) format. This serialized AST serves as a language-agnostic data structure for all subsequent processing stages.

The Framework and Context Analyzer

Following the initial parsing, the **Java** AST is passed to the Framework and Context Analyzer. This component performs static analysis to deduce the technological context of the source code, which is critical for an accurate architectural translation. It inspects the AST for specific Annotation nodes *(e.g., @RestController, @Entity)* to identify the use of frameworks like **Spring or Hibernate**. Concurrently, it examines the project directory structure to locate and parse framework-specific configuration files, such as *struts.xml* for **Apache Struts** or *.gwt.xml* for the **Google Web Toolkit**. The output of this stage is a metadata object that encapsulates the detected frameworks and their configurations, providing essential context to the next phase.

The AI Transformation Core

The AI Transformation Core constitutes the conceptual heart of the system, where the actual translation between language paradigms occurs. This component receives the **Java** AST and the associated framework metadata and formulates a request to a large language model (LLM), specifically **Google's Gemini** model. The model's behavior is governed by an extensive system prompt that encodes a detailed set of rules for transforming **Java** AST nodes into their **Python** AST equivalents. To manage the context window limitations of the LLMs, this component first employs a chunking strategy for large source files. There is a function that is able to divide a class's members into smaller, syntactically complete segments, which are then processed individually by the AI model. The core's output is a JSON object, or a series of JSON objects, that represents the translated **Python** AST.

The Python AST Reconstructor and Code Generator

The final component in the pipeline is responsible for converting the AI-generated **Python** AST from its JSON representation into executable **Python** code. This process begins by parsing the JSON text and recursively instantiating **Python**'s native AST node objects, a task managed by a helper function. If the translation was performed in chunks, this component is also responsible for merging the resulting AST fragments, combining methods into their respective classes, and de duplicating import statements. Once a single, complete, and valid *ast.Module* object for the file has been reconstructed in memory, the built-in *ast.unparse()* function is invoked to generate the final, human-readable **Python** source code, which is then written to a `.py` file. At the same time that this code is being generated, the AI model is also generating multiple unit tests tasked to verify if the **Python** code works and if its logic makes sense. If the sets of tests are all passed it automatically means that the code is correct, but sometimes the tests might not all pass and the generated code is still correct. This can happen due to hallucinations performed by the generative AI model.

6.1.2 Key Technologies and Libraries

The implementation of the translation system is underpinned by a selection of specialized technologies and libraries. Each was chosen to address a specific set of requirements within the pipeline, from initial source code parsing to the final generation of the target language.

ANTLR for Java Parsing

To achieve a reliable and syntactically correct understanding of the source Java code, the system uses a parser generated by Another Tool for Language Recognition. ANTLR is a powerful parser generator that builds a parser from a formal grammar specification. By using a comprehensive, publicly available **Java** grammar, we can ensure that the system is capable of handling the full syntactic complexity of the Java language, including modern language features. The ANTLR-generated parser is invoked via a function that generates the dictionary that stores the AST, which is responsible for transforming the input text into the structured JSON AST representation that the rest of the system consumes. This approach provides a more robust and accurate foundation for analysis than the regular expression-based or manual parsing methods.

Google Gemini for Generative Transformation

The core translation logic, the transformation of **Java** AST into a **Python** AST, is delegated to a generative large language model. The system is specifically configured to use the **Gemini-2.5-flash** model from Google's Gemini family of models. This model was selected for its strong performance in following complex instructions and its ability to adhere to structured data formats such as JSON. The role of LLMs in this architecture is not to translate the code in a conventional text-to-text manner. Instead, it is tasked with a more constrained and structured problem: mapping the nodes and patterns of a source language's AST to the corresponding nodes and patterns of a target language's AST, guided by the extensive set of rules provided in the system prompt. This structured transformation approach leverages the model's pattern recognition capabilities while mitigating the risk of syntactic errors common in direct code generation.

Python's `ast` Module for Code Generation

For all operations related to the target language, the system makes extensive use of **Python**'s built-in *ast* module. This powerful standard library provides a complete toolkit for working with **Python**'s Abstract Syntax Trees programmatically. The `ast` module is central to the final stage of the pipeline. First, the helper function uses the module's classes *(e.g., ast.FunctionDef, ast.Assign)* to convert the LLM-generated JSON into a valid, in-memory **Python** AST. Second, after the full AST for a module has been reconstructed and merged, the *ast.unparse()* function is called. This function traverses the final AST and renders it back into a string of syntactically correct, well-formatted **Python** source code, providing a reliable and safe mechanism for the final code generation step.

6.2 The Translation Pipeline: A Step-by-Step Breakdown

The core of the system is a sequential pipeline that processes each Java file through a series of well-defined phases. This section provides a detailed, step-by-step breakdown of this process, from the initial ingestion of a Java source file to the final generation of its Python equivalent. Each phase builds upon the output of the preceding one, progressively transforming the representation of the code.

6.2.1 Phase 1: Source Code Parsing and AST Generation

The translation process is initiated when a single Java source file (a file with a *.java* extension) is submitted to the pipeline. The first and most foundational phase is the conversion of this raw source text into a structured, hierarchical representation that is amenable to programmatic analysis and transformation.

This critical task is encapsulated within the function in charge of generating the dictionary aimed to store the AST, which is invoked at the beginning of the workflow dedicated to the processing of the original file. This function utilizes a pre-compiled Java parser, which was generated using the Another Tool for Language Recognition (ANTLR) framework from a formal Java grammar. The ANTLR-based parser performs a rigorous lexical and syntactic analysis of the **Java** code, ensuring that its structure is fully understood according to the language's official specification.

The output of the parser is a concrete parse tree, which is subsequently traversed to build a more abstract and simplified representation: the Abstract Syntax Tree. This AST is constructed as a nested structure of **Python** dictionaries, where each dictionary represents a node in the tree *(e.g., a class decla-*

ration, a method call, or a variable assignment) and its key-value pairs represent the node's attributes and children. To facilitate portability and prepare the data for the subsequent AI transformation phase, this dictionary-based AST is serialized into a standardized JavaScript Object Notation string. This JSON object serves as the canonical, language-agnostic representation of the source file for the rest of the pipeline.

6.2.2 Phase 2: Static Analysis for Framework Detection

A purely syntactic translation is insufficient for migrating complex, real-world applications. To achieve a meaningful architectural transformation, the system must understand the high-level frameworks and libraries that govern the application's structure and behavior. Phase 2 is dedicated to this critical task of static analysis, which is orchestrated by a function specialized in identifying any frameworks that might exist in the original **Java**. This function inspects both the AST of the current file and the broader project directory, searching for tell-tale signs of specific technologies.

Detecting Spring and Hibernate via Annotations

Modern **Java** frameworks, particularly **Spring** and **Hibernate** *(or the Java Persistence API, JPA)*, rely heavily on annotations to configure behavior. The system leverages this convention for detection. The *identify_and_map_framework* function recursively traverses the Java AST using the *find_nodes_by_type* helper to locate all `Annotation` nodes. It then inspects the qualified names of these annotations.

The presence of annotations such as `@RestController`, `@Controller`, `@GetMapping`, or `@PostMapping` within a class's AST is a definitive indicator of the **Spring Framework**'s usage for building web services. Similarly, the discovery of annotations like `@Entity`, `@Table`, and `@Id` signals the use of **Hibernate**/**JPA** for Object-relational Mapping (ORM), indicating that the annotated class represents a database entity.

Detecting Struts via `struts.xml` Analysis

Older frameworks like **Apache Struts** often rely on external XML files for configuration rather than annotations. To detect **Struts**, the system shifts its analysis from the AST to the file system. It invokes a function to perform a recursive search of the project directory for a file named *struts.xml*.

If this file is located, it is parsed using **Python**'s standard `xml.etree.ElementTree` library. The system then extracts critical information from the XML structure, specifically the `<action>` mappings.

These mappings define the core logic of a **Struts** application, linking a specific URL path to an action class and a method that should be executed. This information is vital for correctly translating **Struts** actions into modern **Python** web routes.

Detecting GWT via Project Structure

The detection of the **Google Web Toolkit (GWT)** also relies on a file system-based heuristic. The system searches the project structure for the presence of a **GWT** module definition file, which is identified by its *.gwt.xml* extension. The discovery of such a file is a strong indication that the project utilizes **GWT** for its frontend components. Once **GWT** is identified, the analyzer performs a secondary scan of the AST to find interfaces that extend **GWT**'s `RemoteService`, as these define the asynchronous procedure call (RPC) endpoints that need to be migrated to a RESTful API model in **Python**.

Constructing the Framework Metadata Context

The intelligence gathered during this analysis phase is aggregated into a single, structured dictionary named *framework_metadata*. This object contains a list of all detected frameworks *(e.g., ["Spring", "Hibernate"])* and detailed information specific to each, such as the identified Spring endpoints, **Hibernate** entity definitions, or **Struts** action mappings.

This *framework_metadata* dictionary is a critical artifact. It is passed along with the **Java** AST to the AI Transformation Core in the next phase. By providing this explicit context, the system enables the language model to move beyond generic language translation and perform a targeted, framework-aware migration, applying specific rules for converting **Spring** controllers to **Flask** routes or **Hibernate** entities to **SQLAlchemy** models.

6.2.3 Phase 3: AST Chunking for Large Files

A significant practical challenge in designing systems that interact with large language models is managing the finite amount of data the model can process in a single request. Enterprise-scale **Java** files can often be very large, containing numerous fields and methods, which results in a correspondingly large and verbose JSON AST representation. Phase 3 of the pipeline is dedicated to a pre-emptive strategy designed to handle these large files gracefully.

The Challenge of LLM Context Limits

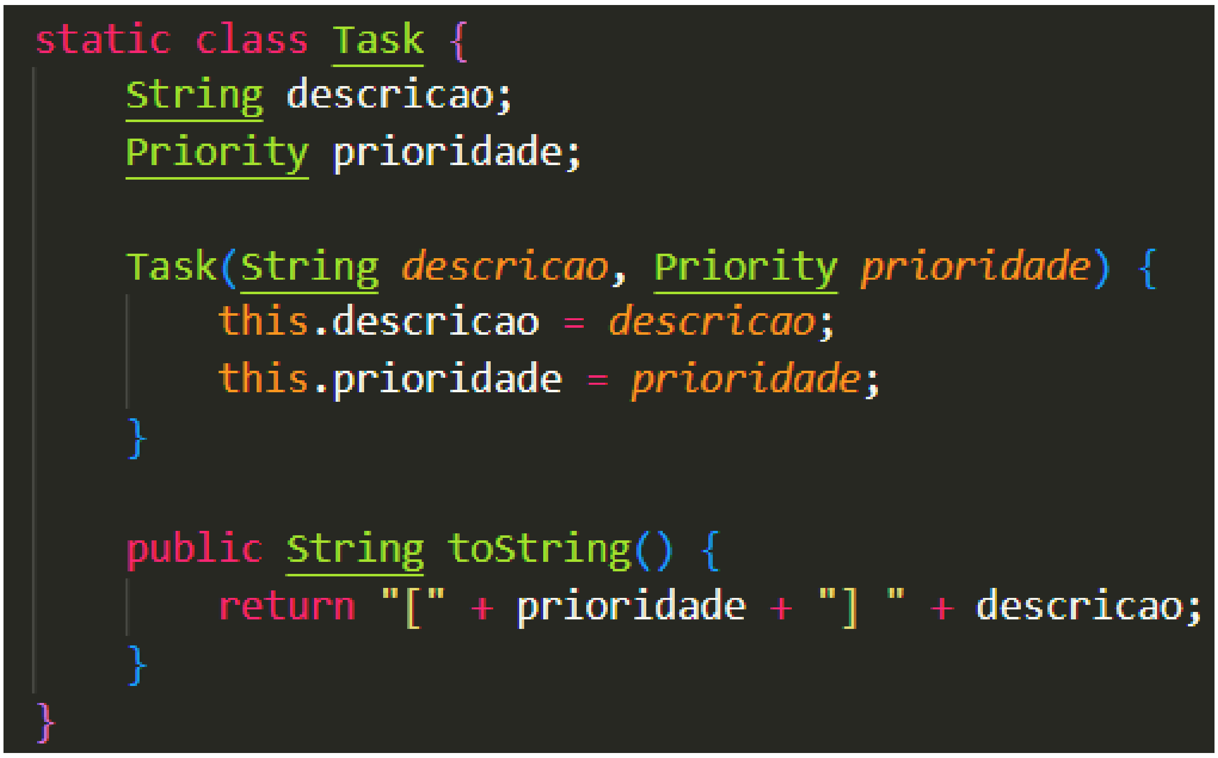
Large language models are constrained by a "context window," which defines the maximum number of tokens (a unit of text, roughly equivalent to a word or part of a word) that can be included in a single prompt and response. Submitting a AST that exceeds this limit would result in an APIs error or, worse, a silently truncated translation and therefore incomplete.

The chunking mechanism was engineered as a robust solution to this fundamental limitation. While the model currently employed in this project, **Gemini 2.5 Flash**, features a very large context window that can accommodate most reasonably sized files without issue, this chunking strategy remains a critical component of the system's design. It ensures forward compatibility with other models that may have more restrictive context limits and guarantees the system's robustness when faced with exceptionally large legacy **Java** files that might still exceed even modern token limits. Therefore, while its necessity may be reduced with the latest models, it is an essential feature for ensuring the tool's reliability and scalability across a wide range of potential inputs and model choices.

Strategy: Splitting by Class Members

The function that splits the AST by class members implements an intelligent, structure-aware splitting strategy. Rather than crudely dividing the JSON text, which would result in syntactically invalid fragments, the function operates on the AST itself.

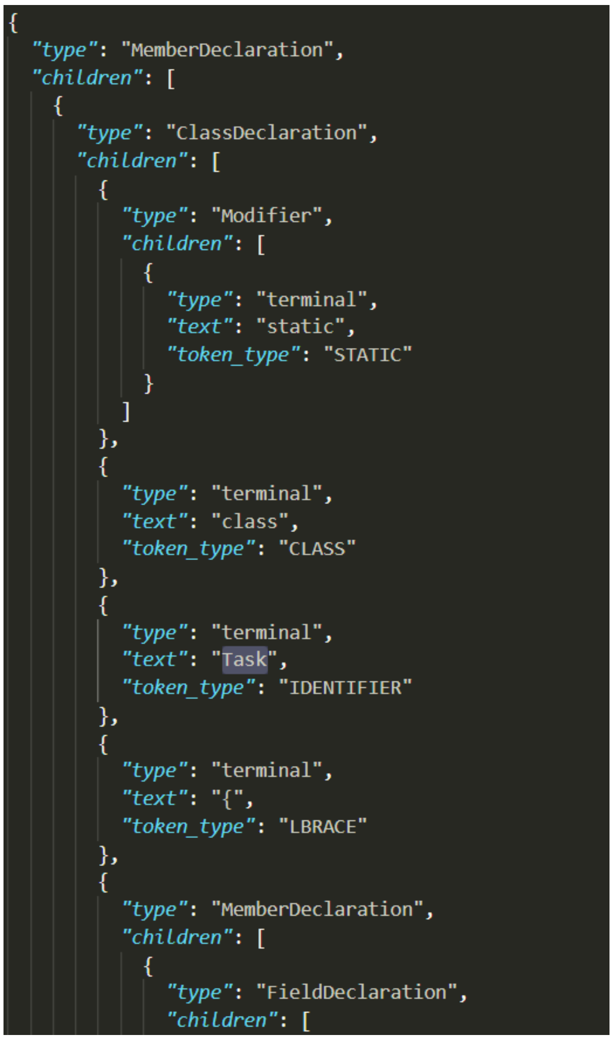
For example, given the following class declaration, shown in Figure 4, the function identifies all the member declarations within its body *(ie, field declarations and method declarations)*. It then groups these members into smaller chunks, with the size of each chunk governed by the *MAX_MEMBERS_PER_CHUNK* constant.



```
static class Task {
    String descricao;
    Priority prioridade;

    Task(String descricao, Priority prioridade) {
        this.descricao = descricao;
        this.prioridade = prioridade;
    }

    public String toString() {
        return "[" + prioridade + "] " + descricao;
    }
}
```

Figure 4: Class Task

Crucially, for each chunk of members, the function reconstructs a complete and syntactically valid `CompilationUnit` `AST`. It does this by creating a copy of the original AST's "scaffolding". This includes the package declaration, all import statements, and the class declaration signature. After that inserts only the members for the current chunk into the class body. The result is a series of smaller, self-contained **Java** ASTs, each of which can be processed independently by the AI Transformation Core without losing the necessary context of imports and class structure. This can be depicted in the image below where the **Class Task** this time is reconstructed as an AST in a JSON format.



```
{
  "type": "MemberDeclaration",
  "children": [
    {
      "type": "ClassDeclaration",
      "children": [
        {
          "type": "Modifier",
          "children": [
            {
              "type": "terminal",
              "text": "static",
              "token_type": "STATIC"
            }
          ]
        },
        {
          "type": "terminal",
          "text": "class",
          "token_type": "CLASS"
        },
        {
          "type": "terminal",
          "text": "Task",
          "token_type": "IDENTIFIER"
        },
        {
          "type": "terminal",
          "text": "{",
          "token_type": "LBRACE"
        },
        {
          "type": "MemberDeclaration",
          "children": [
            {
              "type": "FieldDeclaration",
              "children": [
```

Figure 5: Class Task reconstructed as an AST

6.2.4 Phase 4: AI-Powered AST-to-AST Transformation

This phase represents the conceptual core of the translation system, where the abstract representation of **Java** code is transformed into its **Python** equivalent. Unlike traditional transpilers that rely on a vast set of hand-coded, deterministic rules, this system delegates the complex pattern-matching and transformation logic to a large language model. This approach leverages the model's ability to understand and apply nuanced rules in a flexible manner, allowing for a more sophisticated and context-aware translation.

Prompt Engineering: The Core of the Translation Logic

The success of the AI-powered transformation is almost entirely dependent on the quality and detail of the instructions provided to the model. In this system, this is achieved through a comprehensive **"system prompt"**, a static and detailed set of rules and guidelines that is passed to the model with every **API** call. This prompt, defined in the *system_message* variable, effectively reconfigures the general-purpose language model into a specialized AST-to-AST transpiler. The prompt is meticulously engineered and broken down into several categories of rules.

General Rules and Output Constraints. The most fundamental rules establish the contract for the interaction: *the model's output must be a single, valid JSON object representing a Python AST*. This section of the prompt specifies the required naming conventions for node types *(e.g., Module, FunctionDef)* and their corresponding fields *(e.g., body, targets)*, ensuring that the output conforms strictly to the structure expected by **Python**'s native `ast` module. It explicitly forbids any conversational text, explanations, or extraneous formatting, guaranteeing that the output is machine-parsable.

Framework-Specific Translation Rules. This set of rules enables the system to perform architectural migration, not just syntactic translation. Using the *framework_metadata* provided in the user prompt, the model is instructed to apply specific transformations. For example, it contains rules to convert a **Java** class annotated with **Spring**'s `@RestController` into a series of global **Python** functions decorated with **Flask**'s `@app.route()`. Similarly, it details how to map a **Hibernate** / **JPA** `@Entity` class to a **Python** class inheriting from a **SQLAlchemy** `Base`, translating `@Column` annotations into `sqlalchemy.Column` definitions. This part of the prompt is what allows the system to modernize legacy application architectures.

Idiomatic Python and Type Hinting Rules. To ensure the generated code is not merely a literal translation but is also "Pythonic," the prompt includes rules for idiomatic conversions. For instance, it instructs the model to translate a **Java** null-check like `!myList.isEmpty()` into the more concise and standard **Python** equivalent, `if myList:`. This section also contains critical rules for type hinting, such as the mandatory two-step process for handling potentially null return types: first, adding an import for `Optional` from the `typing` module, and second, wrapping the function's return type annotation in `Optional[...]`.

Critical Syntax Rules. This final category of rules addresses non-negotiable aspects of **Python** syntax that are common pitfalls in language translation. The most important of these is the rule for the `self` parameter: any **Java** method that is not `static` must be translated into a **Python** instance method whose first parameter is explicitly `self`. The prompt provides a template for this `arg` node and emphasizes its mandatory inclusion. It also specifies the exact structure of the *if __name__ == '__main__:* block, ensuring that the translated file has a standard and correct entry point for execution.

Interaction with the Gemini API

The interaction with the language model is a carefully orchestrated process for each AST chunk. A final user prompt is dynamically constructed by combining the static *system_message* with the dynamic data for the current chunk.

The user prompt begins by presenting the framework context, explicitly stating which frameworks were detected by the analyzer in Phase 2 and including the full *framework_metadata* JSON object. This primes the model with the high-level architectural context. Following this, the prompt includes the serialized JSON string of the current **Java** AST chunk. The model is then tasked with generating the corresponding **Python** AST JSON as a response. This entire payload is sent to the **Gemini API** via the *model.generate_content* method, and the system awaits the returned JSON, which represents the successfully transformed AST chunk.

6.2.5 Phase 5: Python AST Reconstruction and Code Generation

The final phase of the pipeline is responsible for converting the abstract, AI-generated representation of the **Python** code into a concrete, syntactically valid source file. This phase takes the JSON output from the AI Transformation Core and meticulously reconstructs a complete, in-memory **Python** AST, which is then used to generate the final code.

Parsing the AI-Generated JSON

The process begins with the raw text response received from the **Gemini API** for each translated chunk. The first step is a sanitization routine that strips any extraneous formatting, such as **Markdown** code fences *(e.g., "'json"')*, that the model might occasionally include. Once cleaned, the raw string is parsed using **Python**'s standard *json.loads()* function, which converts the JSON text into a nested structure of **Python** dictionaries and lists. This dictionary is the working representation of the **Python** AST that will be processed in the subsequent steps.

Converting JSON Dictionaries to Python `ast` Nodes

With the **Python** AST represented as a dictionary, the next step is to convert it into a true in-memory object graph using the classes provided by **Python**'s native `ast` module. This crucial task is performed by an assigned helper function.

This function operates recursively, traversing the dictionary structure. For each dictionary it encounters, it reads the value associated with the `"type"` key (or `"_nodetype"` as a fallback) to determine the corresponding `ast` class that needs to be instantiated *(e.g., ast.FunctionDef, ast.Assign, ast.Call)*. It then dynamically instantiates this class, mapping the keys of the JSON object to the keyword arguments of the class's constructor. This process is applied recursively to all child nodes, effectively "rehydrating" the serialized JSON AST into a live, traversable **Python** `ast` object tree.

Merging AST Chunks

If the original **Java** file was large and consequently split into multiple chunks in Phase 3, the resulting **Python** ASTs must be intelligently recombined into a single, coherent module. The system first collects all the top-level nodes *(e.g., Import, ClassDef, FunctionDef)* from the body of each translated AST chunk into a single list.

It then processes this list to correctly merge class definitions that may have been split. It identifies all methods and attributes belonging to the same class *(e.g., MyClass)* from different chunks and combines them into a single `ClassDef` node for `MyClass`. A final, critical step in this merging process is the de-duplication of import statements. The system iterates through all `Import` and `ImportFrom` nodes, unparses them to a string representation to check for duplicates, and discards any redundant import statements, ensuring the final code is clean and free of unnecessary imports.

Final Code Generation with `ast.unparse`

Once a single, unified `ast.Module` object has been constructed, either from a single chunk or by merging multiple chunks, the final step is to generate the human-readable source code. This is accomplished reliably and safely by invoking the *ast.unparse()* function. This built-in **Python** function traverses the final, complete AST object and renders it back into a string, correctly handling syntax, indentation, and formatting according to **Python**'s language standards. The resulting string is the final output of the translation pipeline, which is then saved to a `.py` file, preserving the original package structure of the source **Java** project.

6.3 Handling Specific Translation Scenarios

This section documents a set of specialized translation rules and strategies implemented in the system to cope with frequent, complex, and framework-specific constructs that appear in real-world **Java** codebases. Each subsection describes the rule set, the motivation, and short examples of the AST transformations (expressed in the familiar `ast`-style notation or short code snippets). Where appropriate the text points to the implementation detail that enforces the rule.

6.3.1 GUI Translation: From Java AWT/Swing to Python Tkinter

Graphical user interfaces written with **Java AWT** or **Swing** have no direct one-to-one equivalence in **Python**. The translator therefore applies a deterministic mapping from common **AWT/Swing** idioms to `tkinter` constructs, producing **Python** code that follows typical **Tkinter** patterns (main window instantiation, widgets with a parent argument, use of layout managers, and event handler refactoring). The concrete mappings and the required AST shapes are encoded in the system prompt and enforced in the reconstruction stage.

Imports and module layout. **AAWT/Swing** import groups such as `java.awt.*` and `java.awt.event.*` are translated into a single **Tkinter** import *(e.g., `import tkinter as tk`)* and, when needed, selective imports for widget classes. The translator ensures these `Import` / `ImportFrom` nodes appear at the top of the module in the correct order (imports first), according to the module ordering rules used in the final code generator.

Window and layout. A `Java Frame` or `JFrame` is mapped to a `tk.Tk()` instance. Size and title calls become `geometry()` and `title()` invocations, and absolute layouts (`setLayout(null)`) are implemented using `place(...)` with explicit `x, y, width, height` arguments. The translator produces the corresponding AST nodes for attribute assignment and method calls so that:

```
frame.setSize(w, h);
frame.setVisible(true);
```

is mapped to:

```
self.f.geometry(f"{w}x{h}")
# later...
self.f.mainloop()
```

and the appropriate `ast.Call` / `ast.Attribute` nodes are emitted.

Widgets and positioning. Widget creation includes the parent as first argument like, in **Java**:

```
Button b = new Button("OK");
b.setBounds(x, y, w, h);
```

becomes **Python**:

```
self.b = tk.Button(self.f, text="OK")
self.b.place(x=x, y=y, width=w, height=h)
```

These are emitted as `ast.Call` / `ast.Attribute` nodes so they integrate cleanly with the **Python** AST reconstruction.

Event handling refactor. **Java** listener patterns such as `b.addActionListener(this)` are refactored to **Tkinter**'s `command=` parameter during widget instantiation. Because **Tkinter** `command` callbacks accept no event object, the translator rewrites **Java** `actionPerformed(ActionEvent e)` handlers into a more explicit dispatcher pattern (one handler method that demultiplexes by widget id or use of `lambda`). The translator generates `lambda` wrapper AST nodes when necessary so widget creation looks like:

```
self.b1 = tk.Button(self.f, text="1", command=lambda: self.
    handle_button_press("b1"))
```

and produces the corresponding:

```
def handle_button_press(self, button_id):
    ...
```

method in the class body.

Edge cases. Calls such as `dispose()` and window-closing hooks are translated to `self.f.destroy()` and `self.f.protocol("WM_DELETE_WINDOW", handler)` respectively. All of the above mappings are deterministic rules in the system prompt and the translator validates the returned **Python** AST fragments against these expectations before reconstruction.

6.3.2 Backend Framework Migration

One of the principal strengths of the system is its framework-aware translation. The framework detector constructs a `framework_metadata` object that captures evidence (annotations, XML files, file structure) and that metadata is fed to the AI core so framework-specific rules can be applied during AST→AST translation. The translator contains explicit rules for **Spring**, **Hibernate**/**JPA**, **Struts** and **GWT**.

Translating Spring MVC to Flask Routes

Detection and intent. Classes annotated with `@RestController` or methods annotated with `@GetMapping` / `@PostMapping` are detected in the static analysis phase and added to `framework_metadata`. The AI transformation is instructed to map these classes and their annotated methods into **Flask** equivalents.

Concrete mapping rules. The translator enforces (via the system prompt) the following mandatory mapping:

- Add `from flask import Flask, jsonify, request` to the module imports and create `app = Flask(__name__)` early in the module body.
- Convert `@GetMapping("/path/id")` into `@app.route("/path/<int:id>", methods=['GET'])` and adapt parameter sources (`@PathVariable`, `@RequestParam`) into function parameters or `request.args` / `request.json` usages.
- Convert `ResponseEntity.ok(payload)` style returns into `return jsonify(payload_dict), 200`.

These rules are encoded in the static system prompt and the final merge phase ensures the generated nodes are correctly ordered (imports, classes, app instantiation, repository instantiation, route functions, main block). *Note: for* **Spring** *controller classes the final combination step deliberately flattens controller method bodies into module-level route functions so that the generated* **Flask** *app follows common* **Flask** *idioms (the implementation flattens controller classes when* **Spring** *is detected).*

Implementation remark. Because this rule changes the structural shape of the code (class methods → module functions), the merging stage contains logic to safely reorder nodes and de-duplicate imports so the produced module does not produce `NameError` exceptions at runtime. The ordering constraints are part of the module assembly in the final code generation step.

Mapping Hibernate/JPA Entities to SQLAlchemy Models

Entity detection and mapping. Classes annotated with `@Entity` (and related annotations) are detected and their field annotations are inspected. The translator maps these into **SQLAlchemy** declarative models: add `Base = declarative_base()` to the module, set `__tablename__` from `@Table(name="...")`, and translate fields annotated with `@Id`/`@Column` into `Column(...)` definitions with appropriate **SQLAlchemy** types and `primary_key` flags. The generated AST will include the necessary **SQLAlchemy** imports *(e.g., `from sqlalchemy import Column, Integer, String`)* at the module top.

Session and query patterns. Typical **Hibernate** `Session` and `EntityManager` patterns (e.g. `session.save(...)` , `createQuery(...)`) are mapped to **SQLAlchemy** session operations (`db_session.add(...)`, `db_session.query(...)`). The translator emits AST nodes for `sessionmaker` initialization and calls to `engine`/`session` as needed. These behaviors are encoded as explicit prompt rules so the AI tends to follow them consistently.

Modernizing Struts Actions and GWT Services into REST APIs

Struts action mapping. If a `struts.xml` is present the analyzer extracts `<action>` mappings and their `result` nodes; those mappings are translated to **Flask** route handlers with the former `authenticate` or `execute` method logic placed inside the route function body. The translator also proposes data validation replacements *(e.g., translating `ActionForm` into Pydantic models or simple dictionaries)* for input handling.

GWT to API + Frontend separation. **GWT RPC** interfaces are translated into REST endpoints and the translator emits a comment placeholders advising that client UI code must be implemented in **JavaScript**. Data Transfer Objects (DTOs) used by **GWT** are emitted as serializable **Python** structures (dictionaries or Pydantic models) so they can be returned by **Flask** endpoints. This architectural shift is explicitly documented in the system prompt because translating **Java** client widgets to **Python** is not feasible.

6.3.3 Core Java-to-Python Language Constructs

Handling the `main` Method and Script Entry Points

The translator recognises `public static void main(String[] args)` and applies a two-part translation:

1. Create a `FunctionDef` named `main` with an `args` parameter typed as `list[str]` (the translator emits the `arguments`/`arg` nodes accordingly)

2. Add an `If` node for the `if __name__ == '__main__':` block that calls the translated `main` (either as `main([])` or `ClassName.main([])` depending on whether `main` was retained as a static method inside a class).

If a Java `main` was translated as a static class method, the translator ensures `@staticmethod` is included in the method's `decorator_list`. These rules are strict because forgetting them would cause runtime errors in Python.

Translating Standard Java Library Classes

Common standard library mappings are implemented as deterministic rules. Examples:

- `Math.sqrt(...)` → `math.sqrt(...)` and an `Import` node for `math`.
- `ArrayList` / `new ArrayList<>()` → **Python** `list literal []`; `add()` → `append()`, `get(i)` → indexing, `size()` → `len(...)`, `isEmpty()` → `not list`.
- `Scanner(System.in)` → translated usage of `input()`; `nextInt()` → `int(input())` (with mention of `ValueError` handling).
- `LocalDateTime.now()` and `DateTimeFormatter.ofPattern(...)` → `datetime.now()` and `strftime(...)` transformations (with `from datetime import datetime` import nodes added).

These conversions are represented by specific `ast` node shapes in the JSON AST the model is asked to produce; the `dict_to_ast_node` rehydration code then instantiates the actual **Python** `ast` nodes used to generate source.

Mapping Project-Local Imports and Class Dependencies

Local **Java** imports of the project *(e.g. `import com.example.model.User;`)* are translated into precise **Python** `ImportFrom` nodes that reference the corresponding module file path *(for example, `from com.example.model.User import User`).* The translator does *not* translate them into `from com.example.model import User` because that treats the class name as a module and leads to runtime type/namespace issues. The correct mapping is enforced by rules in the system prompt and validated when reconstructing the final AST. The output writer also ensures that package directories contain `__init__.py` files where necessary so **Python** imports succeed at runtime.

Addressing Operator and Control Flow Divergences

Differences in language constructs are handled by canonical AST rewrites:

- **Java**'s ++ post-increment is translated into the use of the current value in an expression followed by an explicit `AugAssign` (`+= 1`) on the next statement.
- `switch` statements are converted either to `if/elif/else` chains or to **Python** `match/case` when the pattern is suitable for the newer syntax.
- `try/catch` maps to `try/except` with exception class translations *(e.g., `IllegalArgumentException` → `ValueError` or an appropriate Python equivalent).*
- Ternary expressions (`condition ? a : b`) become conditional expressions (`a if condition else b`).

These translations are both expressed in the model prompt and enforced when validating the AI returned AST fragments before merging.

Chunking and large file handling. A practical, implemented strategy for very large **Java** files is to split a class AST into multiple syntactically complete compilation unit chunks (by class members) so that each chunk fits comfortably in the model context window; each chunk is translated independently and later merged. This chunking algorithm (`split_java_ast_by_class_members`) produces valid `CompilationUnit` fragments that preserve imports and class scaffolding, then after the AI stage the fragments are recombined, their class bodies merged, and imports de-duplicated before final `unparse`. This strategy minimizes prompt errors caused by context limits and preserves correctness across large classes.

Final merging and validation. After all chunks are processed, the pipeline performs a careful merge and ordering pass: it groups imports, merges split class definitions, de-duplicates import statements by unparsing them to string and checking duplicates, and ensures the required module ordering *(imports → classes → app instantiation → repository instantiation → routes → main block)* is respected prior to calling `ast.unparse()` to generate the final **Python** source. This final pass is critical for producing runnable and idiomatic code.

6.4 Tooling and Usability

Beyond its internal architecture, the translation system has been designed and implemented with practical usability in mind. This section describes the aspects of the tool that make it accessible for both experimental and production use, including its command-line interface, flexible file and directory handling, and the structured generation of outputs and logs for traceability.

6.4.1 Command-Line Interface

The core translator is exposed through a simple and intuitive Comand Line Interface (CLI), implemented using **Python**'s `argparse` library. Users can invoke the script by specifying the path to a single **Java** source file or to the root directory of a **Java** project. This input path is parsed from the command line arguments and passed into the `process_java_file` or `process_java_directory` workflows depending on the detected target type.

A typical CLI execution follows the form:

```
python app_automated_ast_translation.py --input path/to/MyClass.java
```

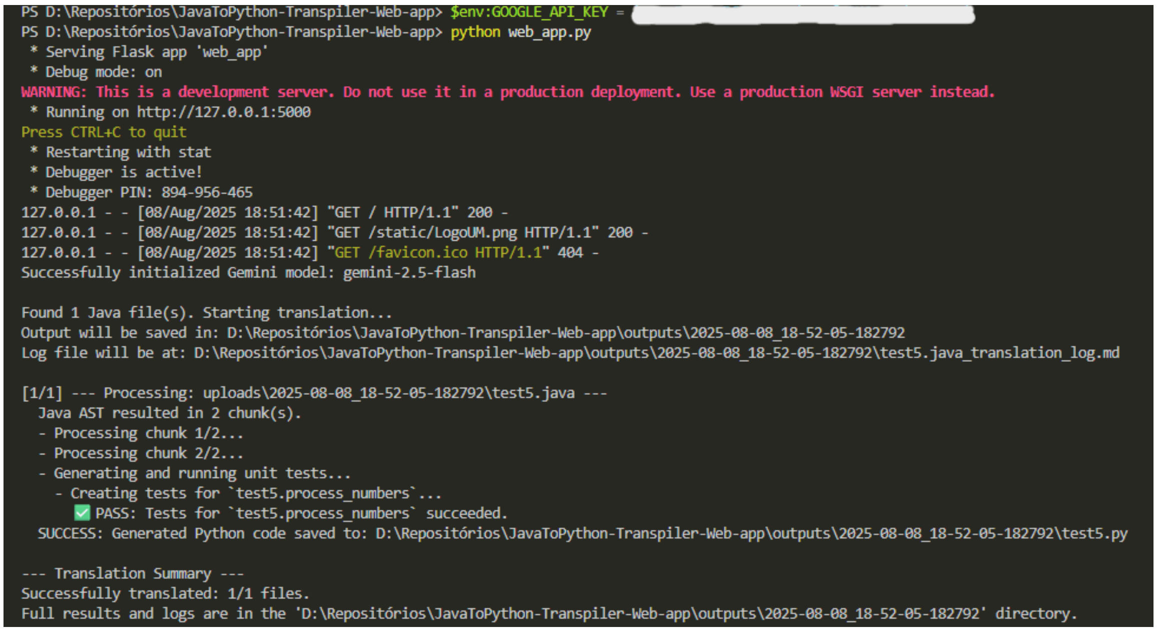
The CLI provides informative console output throughout execution, reporting key milestones such as framework detection, AST chunking, AI translation, and code generation.

In the web application variant (Figure 6), the tool is run indirectly via a **Flask** server endpoint. In this mode, files are uploaded through a browser interface, stored in the `uploads` directory, and then passed to the same core translation pipeline.

6.4.2 File and Directory Handling

The translator can process both individual Java source files and entire project directory trees. In directory mode, the system recursively traverses the given path, locating all files with a `.java` extension. Each file is processed independently through the multi-phase pipeline described in Section 6.2.

For single-file translations, the tool directly reads the file contents, generates the **Java** AST, and passes it into the pipeline. In project mode, the recursive search preserves the relative directory structure so that translated **Python** files maintain the original package layout.



```
PS D:\Repositórios\JavaToPython-Transpiler-Web-app> $env:GOOGLE_API_KEY = [redacted]
PS D:\Repositórios\JavaToPython-Transpiler-Web-app> python web_app.py
 * Serving Flask app 'web_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 894-956-465
127.0.0.1 - - [08/Aug/2025 18:51:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [08/Aug/2025 18:51:42] "GET /static/LogoUM.png HTTP/1.1" 200 -
127.0.0.1 - - [08/Aug/2025 18:51:42] "GET /favicon.ico HTTP/1.1" 404 -
Successfully initialized Gemini model: gemini-2.5-flash

Found 1 Java file(s). Starting translation...
Output will be saved in: D:\Repositórios\JavaToPython-Transpiler-Web-app\outputs\2025-08-08_18-52-05-182792
Log file will be at: D:\Repositórios\JavaToPython-Transpiler-Web-app\outputs\2025-08-08_18-52-05-182792\test5.java_translation_log.md

[1/1] --- Processing: uploads\2025-08-08_18-52-05-182792\test5.java ---
  Java AST resulted in 2 chunk(s).
  - Processing chunk 1/2...
  - Processing chunk 2/2...
  - Generating and running unit tests...
    - Creating tests for `test5.process_numbers`...
      ✅ PASS: Tests for `test5.process_numbers` succeeded.
  SUCCESS: Generated Python code saved to: D:\Repositórios\JavaToPython-Transpiler-Web-app\outputs\2025-08-08_18-52-05-182792\test5.py

--- Translation Summary ---
Successfully translated: 1/1 files.
Full results and logs are in the 'D:\Repositórios\JavaToPython-Transpiler-Web-app\outputs\2025-08-08_18-52-05-182792' directory.
```

Figure 6: Web application mode running locally, showing file processing, chunk handling, and unit test results.

Uploaded files in the web application mode are stored under `uploads/`, grouped into timestamped subdirectories. Generated translations are stored under `outputs/` with matching timestamps, ensuring each translation run is isolated.

6.4.3 Output Generation and Logging

For each translation session, the tool creates a dedicated timestamped output directory. All generated **Python** files are saved here, preserving the original directory and package structure of the source project. In addition to translating the code, the system produces a comprehensive Markdown log file that serves as a detailed audit trail.

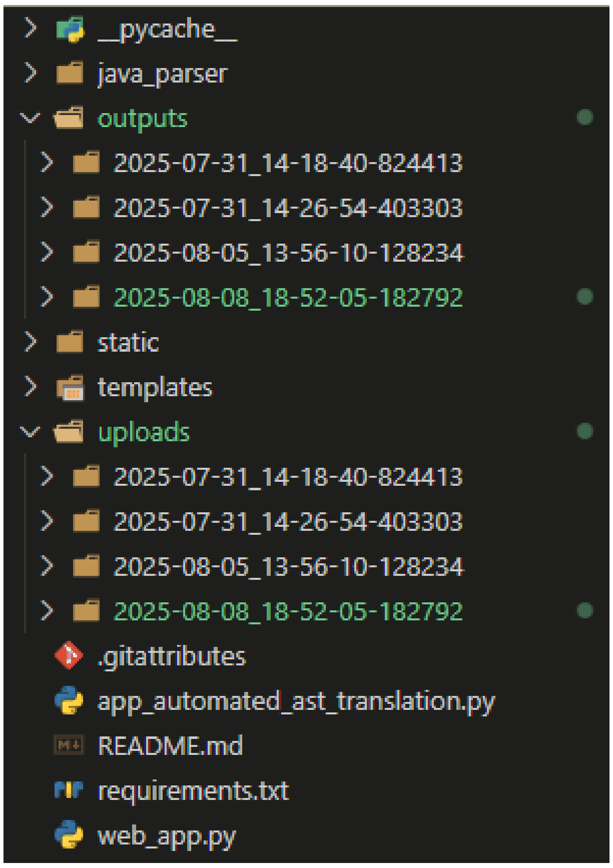
A typical console log during a web session might appear as:

```
Found 1 Java file(s). Starting translation...
Processing chunk 1/2...
Processing chunk 2/2...
PASS: Tests for 'test.process_numbers' succeeded.
SUCCESS: Generated Python code saved to outputs/2025-08-08_18-52-05/test
    .py
```

The log file records:

- Framework detection results.
- Details of AST chunking for large files.
- AI model interactions and responses.
- Unit test generation and pass/fail status.
- Any warnings or errors encountered.

Figure 7 illustrate these aspects in practice, showing the resulting project directory layout.



```
> __pycache__
> java_parser
v outputs
  > 2025-07-31_14-18-40-824413
  > 2025-07-31_14-26-54-403303
  > 2025-08-05_13-56-10-128234
  > 2025-08-08_18-52-05-182792
> static
> templates
v uploads
  > 2025-07-31_14-18-40-824413
  > 2025-07-31_14-26-54-403303
  > 2025-08-05_13-56-10-128234
  > 2025-08-08_18-52-05-182792
  .gitattributes
  app_automated_ast_translation.py
  README.md
  requirements.txt
  web_app.py
```

Figure 7: Project folder structure with timestamped `uploads` and `outputs` directories.

6.5 Web App Integration

To enhance the usability and accessibility of the translation system, the core command-line functionality was encapsulated within a web application. This provides a graphical user interface (GUI) that allows users to interact with the tool without needing to operate in a terminal environment, thus making the system available to a broader audience. This section details the architecture of this web application and the design of its user interface.

6.5.1 Converting the App to a Web Application

The web application was developed using **Flask**, a lightweight and flexible web framework for **Python**. The choice of **Flask** was motivated by its simplicity and the ease with which it can be integrated with existing **Python** code. The *web_app.py* script serves as the entry point for the web server and orchestrates the interaction between the user and the core translation engine.

A key design principle was the separation of concerns between the web interface and the translation logic. The core translation functionality, contained within *app_automated_ast_translation.py*, is imported into the web application as a single function, *run_translation*. The web application is responsible for handling HTTP requests, managing file uploads, and rendering **HTML** templates, while the imported function remains solely responsible for executing the translation pipeline.

The process is initiated via the `/transpile` endpoint, which accepts `POST` requests from the main web form. This endpoint performs the following steps:

1. Creates a unique directory for the current request to isolate its files from other concurrent requests.
2. It determines whether the user is uploading a single `.java` file or a `.zip` archive containing a full project.
3. The uploaded file is saved to the unique request directory. If it is a `.zip` archive, its contents are extracted.
4. The *run_translation* function is then invoked, with the path to the uploaded source file or directory passed as an argument. Critically, an *output_root_dir_override* argument is also passed, directing the translation engine to place its output into the request-specific directory.
5. Upon completion, the **Flask** application reads the generated **Python** files and the **Markdown** log file from the output directory.
6. Finally, it renders the `result.html` template, passing the translated code, log content, and other contextual information to the user's browser.

This architecture effectively wraps the powerful command-line tool in a user-friendly web service, handling all the temporary file management and process invocation behind the scenes.

6.5.2 Web App Interface

The user interface is designed to be clean, intuitive and informative, guiding the user through the translation process and presenting the results in a clear and organized manner. It consists of two primary views: the upload page and the results page.

The main page of the application, shown in Figure 8, presents the user with a straightforward choice: translating a single file or an entire project. The interface dynamically updates to show the appropriate file input control based on the user's selection, minimizing clutter and preventing user error.

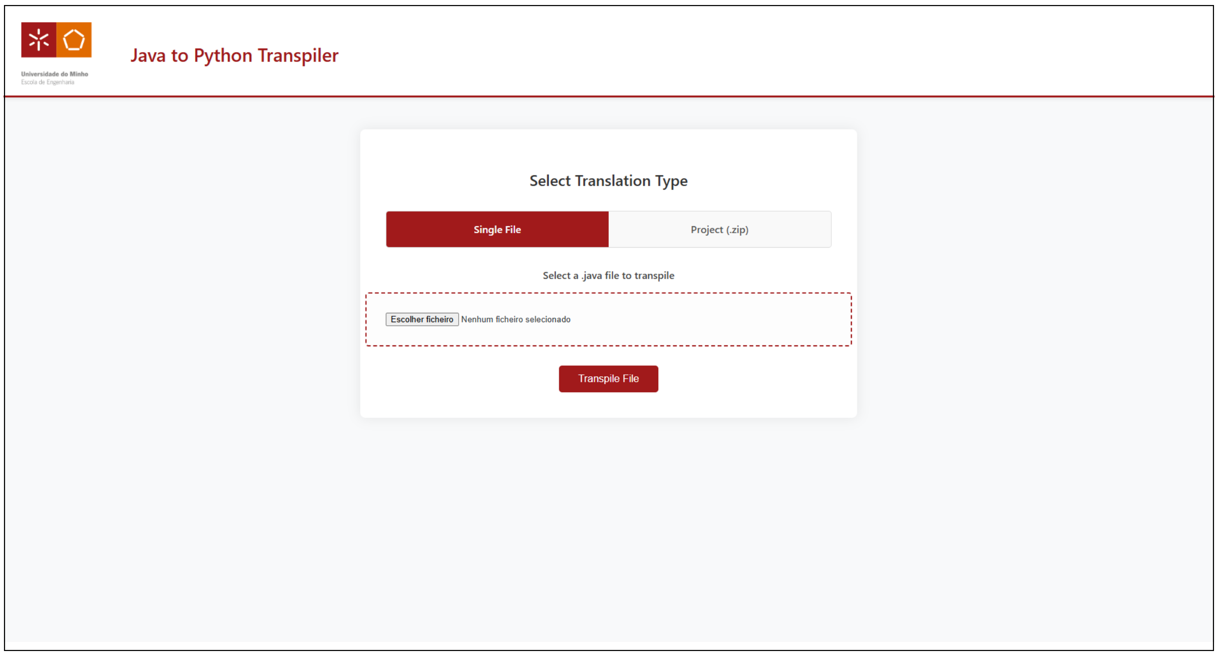
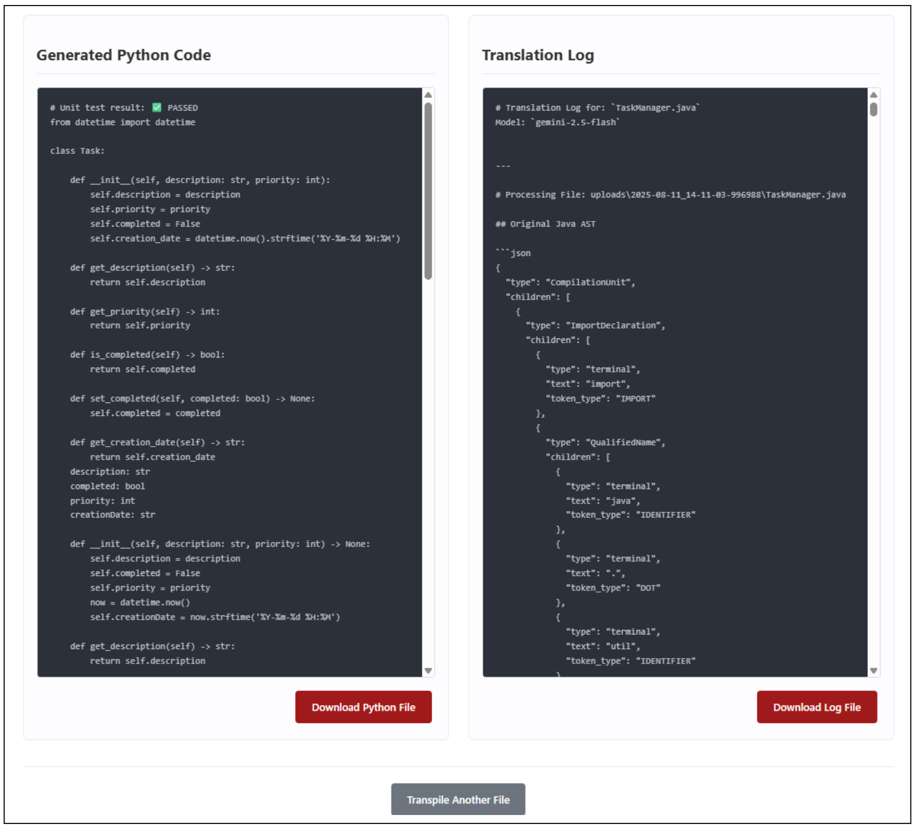


Figure 8: The main upload interface of the web application.

After the translation is complete, the user is redirected to a results page, which intelligently adapts its layout based on the type of translation performed. For a single-file translation, the interface is split into a two-panel view, as depicted in Figure 9. The left panel displays the generated **Python** code, while the right panel shows the detailed translation log. This allows for a direct comparison and review of the output. Download buttons are provided for both the code and the log file.

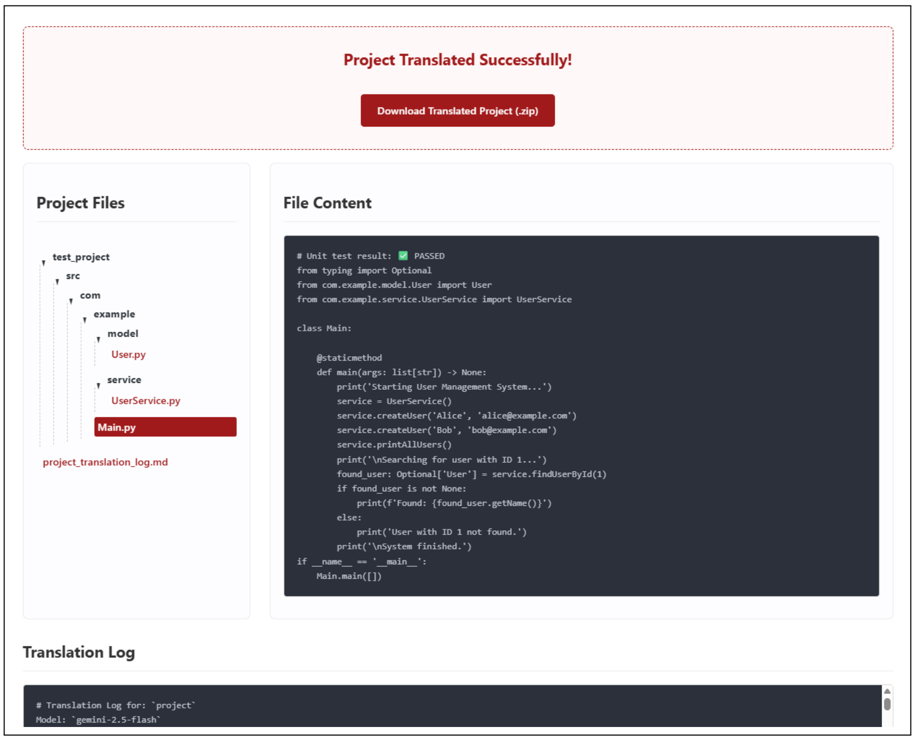
For a multi-file project translation, the results page adopts a more comprehensive layout suited for navigating a directory structure, as shown in Figure 10. A prominent button at the top allows the user to download the entire translated project as a single `.zip` archive. Below this, the main view consists of an interactive file tree on the left and a content viewer on the right. The user can browse the translated project's directory structure and click on any file to view its contents, which are fetched and displayed asynchronously. The full translation log is also presented at the bottom of the page for a complete overview of the entire project's translation process.



```
# Unit test result: ✅ PASSED
from datetime import datetime

class Task:

    def __init__(self, description: str, priority: int):
        self.description = description
        self.priority = priority
        self.completed = False
        self.creation_date = datetime.now().strftime('%Y-%m-%d %H:%M')

    def get_description(self) -> str:
        return self.description

    def get_priority(self) -> int:
        return self.priority

    def is_completed(self) -> bool:
        return self.completed

    def set_completed(self, completed: bool) -> None:
        self.completed = completed

    def get_creation_date(self) -> str:
        return self.creation_date
    description: str
    completed: bool
    priority: int
    creationDate: str

    def __init__(self, description: str, priority: int) -> None:
        self.description = description
        self.completed = False
        self.priority = priority
        now = datetime.now()
        self.creationDate = now.strftime('%Y-%m-%d %H:%M')

    def get_description(self) -> str:
        return self.description
```

~~~
# Translation Log for: `TaskManager.java`
Model: `gemini-2.5-flash`


---

# Processing File: uploads\2025-08-11_14-11-03-996988\TaskManager.java

## Original Java AST

```json
{
  "type": "CompilationUnit",
  "children": [
    {
      "type": "ImportDeclaration",
      "children": [
        {
          "type": "terminal",
          "text": "import",
          "token_type": "IMPORT"
        },
        {
          "type": "QualifiedName",
          "children": [
            {
              "type": "terminal",
              "text": "java",
              "token_type": "IDENTIFIER"
            },
            {
              "type": "terminal",
              "text": ".",
              "token_type": "DOT"
            },
            {
              "type": "terminal",
              "text": "util",
              "token_type": "IDENTIFIER"
~~~

Figure 9: The results page for a single-file translation.



```
# Unit test result: ✅ PASSED
from typing import Optional
from com.example.model.User import User
from com.example.service.UserService import UserService

class Main:

    @staticmethod
    def main(args: list[str]) -> None:
        print('Starting User Management System...')
        service = UserService()
        service.createUser('Alice', 'alice@example.com')
        service.createUser('Bob', 'bob@example.com')
        service.printAllUsers()
        print('\nSearching for user with ID 1...')
        found_user: Optional['User'] = service.findUserById(1)
        if found_user is not None:
            print(f'Found: {found_user.getName()}')
        else:
            print('User with ID 1 not found.')
        print('\nSystem finished.')
if __name__ == '__main__':
    Main.main([])
```

```
# Translation Log for: `project`
Model: `gemini-2.5-flash`
```

Figure 10: The results page for a multi-file translation.

6.6 Chapter Summary

This chapter has provided a detailed technical exposition of an automated **Java-to-Python** translation system. The core of this work is a novel approach that moves beyond traditional lexical transpilation by operating on a higher level of abstraction: the Abstract Syntax Tree (AST). The system's architecture is designed as a modular, multi-phase pipeline that systematically transforms **Java** source code into modern, idiomatic **Python**.

The process begins with the **Java Parser Frontend**, which uses ANTLR to create a robust JSON representation of the source code's AST. This is followed by the **Framework and Context Analyzer**, a static analysis phase that inspects the code and project structure to detect the use of major frameworks like **Spring**, **Hibernate**, **Struts**, and **GWT**, compiling these findings into a critical metadata object.

The central component is the **AI Transformation Core**, which leverages the **Google Gemini** language model. Guided by a meticulously engineered system prompt containing hundreds of specific rules, the AI transforms the **Java** AST into an equivalent **Python** AST. This prompt-driven approach allows for sophisticated, context-aware translations, including architectural migrations *(E.G., SPRING TO FLASK)* and the handling of complex language idioms. To ensure reliability, a chunking mechanism was implemented to manage large source files by splitting and processing them in segments.

The final phase, the **Python AST Reconstructor and Code Generator**, takes the AI's JSON output, reconstructs a valid in-memory **Python** AST using the native `ast` module, merges any translated chunks, and unparses the final tree into executable **Python** code. The system's effectiveness was detailed through its handling of specific scenarios, from GUI and backend framework migrations to the precise translation of core language constructs.

Finally, the chapter described the system's practical usability, manifested as both a command-line tool and an intuitive **Flask** web application. This dual interface makes the powerful translation engine accessible to a wide range of users, demonstrating a complete and well-rounded implementation of an advanced, AI-augmented code translation solution.

To round up this chapter here is a list of everything developed in this project:

- The application implemented **(JavaToPython)**: http://207.154.252.171/

- Website created about the paper published, it includes the document, the tools tested and the scripts used to evaluate the output precision: https://justandre02.github.io/Comparative-Review-and-Empirical-Evaluation-about-transpilers/
- Landing page abou the project that redirects the user to the things mentioned before as well as the document you are reading: http://www.java.2.python.pt/

Chapter 7

Conclusion

Each chapter is designed to build upon the previous one, creating a coherent narrative that systematically addresses the research objectives. The document progresses from theoretical foundations through empirical investigation to critical analysis and conclusive insights.

Introduction

The initial chapter establishes the research context, presenting motivation, objectives, and the fundamental challenges in code translation. It provides a critical overview of the intersection between transpilation techniques and machine learning approaches.

State of the Art

This section provides an in-depth analysis of the state of the art in code translation, beginning with a comprehensive examination of transpilers and progressing towards the emerging integration of Artificial Intelligence in this field. It first explores the theoretical foundations of transpilation, outlining the formal principles that govern source-to-source code conversion and the historical evolution that led from early, rule-based tools to more advanced and modular architectures. The technical mechanics behind these systems are discussed in detail, describing how they rely on parsing, Abstract Syntax Tree (AST), and transformation rules to achieve accurate translation between programming languages. In addition, the typological classifications of transpilers are reviewed, distinguishing between language-specific implementations, intermediate-representation-based systems, and hybrid approaches. The analysis further considers practical applications—including legacy code migration, platform interoperability, and performance optimization—before identifying persistent challenges and limitations, such as semantic drift, framework dependencies, and the substantial development effort required to maintain precision across diverse languages.

Building upon this foundation, the chapter then examines the role of Artificial Intelligence (AI) in code translation, highlighting how advancements in Machine Learning (ML) and Natural Language Processing (NLP) have transformed the landscape of program transformation. It begins by presenting the core AI concepts relevant to this domain, emphasizing how generative and transformer-based models learn to represent both the syntax and semantics of code. The discussion extends to techniques for automated code analysis and transformation, exploring methods for error detection, refactoring, and optimization driven by data-centric learning. Moreover, it investigates strategies for cross-linguistic code translation, showing how AI systems can capture contextual intent beyond direct syntactic correspondence. The section concludes by considering practical applications and performance impacts of AI-assisted translation and reflecting on future directions and emerging challenges, including scalability, interpretability, and maintaining semantic fidelity in increasingly complex translation scenarios.

Project Proposal

This chapter presented a comprehensive proposal for a hybrid code translation system that combines traditional transpilation techniques with modern machine learning approaches. Through detailed sequence diagrams and component analysis, it was explained how each element of the system contributes to the overall goal of accurate and efficient code translation while maintaining code integrity and functionality.

Research conducted about transpilers

In this chapter an independent comparative analysis of various code transpilers was conducted. The research aim to compare all the transpilation tools focused on four main metrics: execution time, CPU usage, memory consumption, and translation accuracy.

Implementation

In this chapter, every aspect of the application developed in this project is documented. This means contextualizing the application, explaining the different phases of the application with snippets of code and the inclusion of images depicting the system outcomes.

7.1 Results

In this section, the main outcomes of the system developed will be emphasized. Chapter 5 discussed the study carried out to test systematically several existing transpilers. Following precisely the same approach, the application developed was also tested, using the same input code in order to have a direct comparison to the results gathered in the previous research.

With that being said, the application developed showed the following performance results:

- **Execution time:** 0,67 seconds
- **CPU Usage:** less than 1%
- **RAM Usage:** 92,66 MB

Concerning the precision of the output code this application registered a **95 out of 100 score**, using the same benchmark used for the **Java2Python** test.

It is important to note that, although the application reported an execution time of less than one second, the tool actually required around 20 seconds to complete its processing. This difference occurs because the user's machine handles most local tasks quickly, while the main operations depend on a generative AI model. This process includes tasks like AST translation, AST splitting, member chunking and generation of unit tests to the AST translated. This process may seem a lit bit redundant and the sequence of tasks performed causes the high execution time but that approach is essential to the project architecture, ensuring an extra layer of security and precision.

When compared directly with the **Java2Python** tool, the results may seem disappointing. However, this applies mainly to small examples. In larger projects, although **Java2Python** remains faster, it requires much more computational power and produces many more errors in the translated code. Another issue that distinguishes both applications is the fact that the developed application is built to also translate whole projects, which is much more useful to companies and users in general who want to use this application to take past work done and transform it.

7.2 Final Thoughts

This project has been a rewarding experience. All the objectives initially proposed were successfully achieved. Additional contributions, such as the publication of an article[1], increase the author's satisfaction.

[1] https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.SLATE.2025.11

Summing up, it has been demonstrated that **the integration of Machine Learning techniques into transpilation processes significantly enhances their overall effectiveness**. Naturally, there are minor drawbacks, such as the occasional variation in the output generated by the AI model when given the exact same prompt. However, this aspect was thoroughly analyzed during the project, and it was observed that even when the model produced different outputs, the results remained consistently correct, albeit with some variations in detail.

In today's AI landscape, this project is highly significant, showing how AI can address programming challenges that are often neglected. While this work focused on the translation of **Java** into **Python**, two of the most widely used programming languages, the same methodology could be applied to many others.

7.3 Future Work

In the recent future it would be interesting to return to this project in order to improve some smaller aspects that still make a lot of difference. One of them is, for example, the section where the ASTs are tested. Although that section is working it has a few inconsistencies created by hallucinations made by the AI model. It would be ideal to find a reliable solution in order to end this problem. Another future task would be upgrading the framework detection side of the application in order to have direct switching between languages in more frameworks from **Java** instead of only the five most popular. This would ensure that the generative AI model doesn't choose wrong frameworks when task to translate a file or project that involves a framework.

Looking ahead into a more distant future, one natural continuation of this research would involve extending the approach to legacy programming languages, such as **COBOL**, and translating them into modern equivalents. This is of particular importance considering that such languages continue to be widely used in the job market, especially within the financial sector. In the past year alone, more than 4,000 job postings in the United States required expertise in these technologies.

If this project left you interested, check out the website[2] developed associated with this thesis, where you can find the document you just read,a redirection button that takes you to the application developed, and also another website related to the paper published about this whole project.

[2] http://www.java.2.python.pt/

Bibliography

Duaa Alawad, Manisha Panta, Minhaz Zibran, and Md Rakibul Islam. An empirical study of the relationships between code readability and software complexity, 2019. URL https://arxiv.org/abs/1909.01760.

Miltiadis Allamanis. *Graph Neural Networks in Program Analysis*, pages 483–497. Springer Nature Singapore, Singapore, 2022. ISBN 978-981-16-6054-2. doi: 10.1007/978-981-16-6054-2_22. URL https://doi.org/10.1007/978-981-16-6054-2_22.

Benjamin Allen. *COBOL: The Pentagon, United States of America, 1959*, pages 37–45. 09 2024. ISBN 9781531506629. doi: 10.5422/fordham/9781531506629.003.0004.

Tiago L Alves, Paulo F Silva, and Joost Visser. Constraint-aware schema transformation. *Electronic Notes in Theoretical Computer Science*, 290:3–18, 2012.

Bastidas F. Andrés and María Pérez. Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017. doi: 10.1109/ETCM.2017.8247456.

Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of uml/ocl refactoring rules. In *Perspectives of Systems Informatics: 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers 6*, pages 70–83. Springer, 2007.

Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502, 2020.

Andrés Bastidas Fuertes, María Pérez, and Jaime Meza. Transpiler-based architecture design model for back-end layers in software development. *Applied Sciences*, 13(20):11371, 2023.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/17c3433fecc21b57000debdf7ad5c930-Paper.pdf.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

Amel Bennaceur and Karl Meinke. *Machine Learning for Software Analysis: Models, Methods, and Applications*, pages 3–49. Springer International Publishing, Cham, 2018. ISBN 978-3-319-96562-8. doi: 10.1007/978-3-319-96562-8_1. URL https://doi.org/10.1007/978-3-319-96562-8_1.

Victor Berdonosov and Alena Zhivotova. The evolution of the object-oriented programming languages. *Scholarly Notes of Komsomolsk-na-Amure State Technical University*, 1:35–43, 06 2014. doi: 10.17084/2014.II-1(18).5.

Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.38. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.38.

Saikat Chakraborty and Baishakhi Ray. On multi-modal learning of editing source code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 443–455, 2021. doi: 10.1109/ASE51524.2021.9678559.

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.*, 34(7):1–9, May 1999. ISSN 0362-1340. doi: 10.1145/315253.314414. URL https://doi.org/10.1145/315253.314414.

Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2008.02.068. URL https://www.sciencedirect.com/science/article/pii/S0164121208000502. Best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007.

Michael Courtney, Michael Breen, Iain McMenamin, and Gemma McNulty. Automatic translation, context, and supervised learning in comparative politics. *Journal of Information Technology & Politics*, 17(3): 208–217, 2020.

A. Cox and C. Clarke. Syntactic approximation using iterative lexical analysis. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 154–163, 2003. doi: 10.1109/WPC.2003.1199199.

Hua Fang, Meng Wang, Gushu Li, Bo Peng, Chenxu Liu, Mao Zheng, S.R. Stein, Yufei Ding, Eddy Z. Zhang, Travis S. Humble, and Ang Li. Qasmtrans: A qasm based quantum transpiler framework for nisq devices, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL https://arxiv.org/abs/2002.08155.

W.T. Freeman. Computer vision for television and games. In *Proceedings International Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems. In Conjunction with ICCV'99 (Cat. No.PR00378)*, pages 118–118, 1999. doi: 10.1109/RATFG.1999.799233.

Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. Transpilers: A systematic mapping review of their usage in research and industry. *Applied Sciences*, 13(6):3667–3667, 2023. doi: 10.3390/app13063667.

Nadja Gaudillière-Jami. AD Magazine: Mirroring the Development of the Computational Field in Architecture 1965–2020. In *ACADIA 2020: Distributed Proximities*, volume 1, pages 150–159, Online, France, October 2020. B. Slocum, V. Ago, S. Doyle, A. Marcus, M. Yablonina, M. del Campo. URL https://hal.science/hal-04588619.

Joshua Goodman. Parsing algorithms and metrics. 34, 04 1999. doi: 10.3115/981863.981887.

Gregory Grefenstette. *Tokenization*, pages 117–133. Springer Netherlands, Dordrecht, 1999. ISBN 978-94-015-9273-4. doi: 10.1007/978-94-015-9273-4_9. URL https://doi.org/10.1007/978-94-015-9273-4_9.

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems*, 37, 07 2015. doi: 10.1145/2743016.

Dick Grune and Ceriel J. H. Jacobs. *Introduction to Parsing*, page 61–102. Springer New York, 2008. ISBN 9780387689548. doi: 10.1007/978-0-387-68954-8_3. URL http://dx.doi.org/10.1007/978-0-387-68954-8_3.

Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 601–612, 2022. doi: 10.1109/SANER53432.2022.00077.

Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 930–937, 2019.

Michael T Heath. *Scientific computing: an introductory survey, revised second edition*. SIAM, 2018.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051.

Jean-Michel Hoc and Anh Nguyen-Xuan. Chapter 2.3 - language semantics, mental models and analogy. In J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 139–156. Academic Press, London, 1990. ISBN 978-0-12-350772-3. doi: https://doi.org/10.1016/B978-0-12-350772-3.50014-8. URL https://www.sciencedirect.com/science/article/pii/B9780123507723500148.

Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, Wang Gao, and Mengting Yuan. Unsupervised software repositories mining and its application to code search. *Software: Practice and Experience*, 50(3):299–322, 2020. doi: https://doi.org/10.1002/spe.2760. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2760.

Evgeniy Ilyushin and Dmitry Namiot. On source-to-source compilers. *International Journal of Open Information Technologies*, 4(5):48–51, 2016.

Philip Japikse, Kevin Grossnicklaus, and Ben Dewey. *Introduction to TypeScript*, pages 413–468. Springer, 12 2019. ISBN 978-1-4842-5351-9. doi: 10.1007/978-1-4842-5352-6_10.

Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An ast structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, PP:1–1, 12 2021. doi: 10.1109/TASLP.2021.3138717.

Bo Jin. Neural machine translation based on semantic word replacement. In *Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security*, GAIIS '24, page 106–112, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400709562. doi: 10.1145/3665348.3665368. URL https://doi.org/10.1145/3665348.3665368.

A. Johnson. Fortran preprocessors. *Computer Physics Communications*, 45:275–281, 08 1987. doi: 10.1016/0010-4655(87)90164-0.

Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating glr parsing algorithms. *Science of Computer Programming*, 61:228–244, 08 2006. doi: 10.1016/j.scico.2006.04.004.

Aravind K. Joshi. Natural language processing. *Science*, 253(5025):1242–1249, 1991. doi: 10.1126/science.253.5025.1242. URL https://www.science.org/doi/abs/10.1126/science.253.5025.1242.

Yoon Kim. Sequence-to-sequence learning with latent neural grammars. *ArXiv*, abs/2109.01135, 2021. URL https://api.semanticscholar.org/CorpusID:237385685.

Philipp Koehn and Rebecca Knowles. Six challenges for neural machine translation. *arXiv preprint arXiv:1706.03872*, 2017.

Kostadin Kratchanov and Efe Ergün. Language interoperability in control network programming, 2018.

Victoria Kripets. Machine translation in banking and financial operations. https://lingvanex.com/blog/machine-translation-in-banking-and-financial-operations/, 2024. Accessed: 2025-02-27.

Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2006.10.017. URL https://www.sciencedirect.com/science/article/pii/S0950584906001820. 12th Working Conference on Reverse Engineering.

Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.*, 47(10):147–162, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384628. URL https://doi.org/10.1145/2398857.2384628.

Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14967–14979. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/7d6548bdc0082aacc950ed35e91fcccb-Paper.pdf.

Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 1. deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys*, 2020. doi: 10.1145/3383458.

Hugh Leather and Chris Cummins. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8, 2020. doi: 10.1109/FDL50818.2020.9232934.

Celine Lee, Justin Gottschlich, and Dan Roth. Toward code generation: A survey and lessons from semantic parsing, 2021. URL https://arxiv.org/abs/2105.03317.

Wenke Lee, Salvatore J Stolfo, and Philip K Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI workshop on AI approaches to fraud detection and risk management*, pages 50–56. New York;, 1997.

Zheng Li, Yonghao Wu, Bin Peng, Xiang Chen, Zeyu Sun, Yong Liu, and Doyle Paul. Setransformer: A transformer-based code semantic parser for code comment generation. *IEEE Transactions on Reliability*, 72(1):258–273, 2023. doi: 10.1109/TR.2022.3154773.

P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006. doi: 10.1109/MS.2006.114.

Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. Embedding api dependency graph for neural code generation. *Empirical Software Engineering*, 26:1–51, 2021.

Chenyang Lyu, Zefeng Du, Jitao Xu, Yitao Duan, Minghao Wu, Teresa Lynn, Alham Fikri Aji, Derek F Wong, Siyou Liu, and Longyue Wang. A paradigm shift: The future of machine translation lies with large language models. *arXiv preprint arXiv:2305.01181*, 2023.

Machinet. The evolution of automated code generation tools: Advancements and best practices. https://blog.machinet.net/post/the-evolution-of-automated-code-generation-tools-advancements-and-best-practices?utm_source=chatgpt.com, 2023. Accessed: 2025-02-27.

Dan Maharry. *TypeScript revealed*. Apress, 2013.

Arun Kumar Marandi and Danish Ali Khan. An impact of linear regression models for improving the software quality with estimated cost. *Procedia Computer Science*, 54:335–342, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.06.039. URL https://www.sciencedirect.com/science/article/pii/S1877050915013630. Eleventh International Conference on Communication Networks, ICCN 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Data Mining and Warehousing, ICDMW 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Image and Signal Processing, ICISP 2015, August 21-23, 2015, Bangalore, India.

André Melo, Nathan Earnest-Noble, and Francesco Tacchino. Pulse-efficient quantum machine learning. *Quantum*, 7:1130, 2023.

Rada Mihalcea, Hugo Liu, and Henry Lieberman. Nlp (natural language processing) for nlp (natural language programming). In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, pages 319–330, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32206-1.

MinéAntoine. 3. field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. *Sigplan Notices*, 2006. doi: 10.1145/1159974.1134659.

Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 234–242, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568324. URL https://doi.org/10.1145/2568225.2568324.

Martin Monperrus. Explainable software bot contributions: Case study of automated bug fixes. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 12–15, 2019. doi: 10.1109/BotSE.2019.00010.

Dolly M Neumann. Evolution process for legacy system transformation. In *IEEE Technical Applications Conference. Northcon/96. Conference Record*, pages 57–62. IEEE, 1996.

Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, pages 1–12, 2023. doi: 10.1109/ms.2023.3243858.

Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, 41(1):105–112, 2024. doi: 10.1109/MS.2023.3243858.

Indranil Palit and Tushar Sharma. Generating refactored code accurately using reinforcement learning, 2024. URL https://arxiv.org/abs/2412.18035.

Katerina Paltoglou, Vassilis E. Zafeiris, N.A. Diamantidis, and E.A. Giakoumakis. Automated refactoring of legacy javascript code to es6 modules. *Journal of Systems and Software*, 181:111049, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111049. URL https://www.sciencedirect.com/science/article/pii/S0164121221001461.

Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. 24(3). ISSN 0742-1222, 1557-928X. doi: 10.2753/MIS0742-1222240302.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023. URL https://arxiv.org/abs/2302.06590.

Ben Peters, Vlad Niculae, and André F. T. Martins. Sparse sequence-to-sequence models, 2019. URL https://arxiv.org/abs/1905.05702.

Alberto Pettorossi, Maurizio Proietti, Fabio Fioravanti, and Emanuele De Angelis. A historical perspective on program transformation and recent developments (invited contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM 2024, page 16–38, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704871. doi: 10.1145/3635800.3637446. URL https://doi.org/10.1145/3635800.3637446.

David A Plaisted. Source-to-source translation and software engineering. 2013.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. 2017. doi: 10.18653/V1/P17-1105.

Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018. URL https://api.semanticscholar.org/CorpusID:49313245.

Hardik Raina, Aditya Kurele, Nikhil Balotra, and Krishna Asawa. Language agnostic neural machine translation. In *Proceedings of the 2024 Sixteenth International Conference on Contemporary Computing*, IC3-2024, page 535–539, New York, NY, USA, 2024. Association for Computing Machinery.

ISBN 9798400709722. doi: 10.1145/3675888.3676109. URL https://doi.org/10.1145/3675888.3676109.

Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Programming reconfigurable heterogeneous computing clusters using mpi with transpilation. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 1–9, Nov 2020. doi: 10.1109/H2RC51942.2020.00006.

Ruslan Salakhutdinov. Deep learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 1973, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329569. doi: 10.1145/2623330.2630809. URL https://doi.org/10.1145/2623330.2630809.

Larissa Schneider and Dominik Schultes. Evaluating swift-to-kotlin and kotlin-to-swift transpilers. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '22, page 102–106, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393010. doi: 10.1145/3524613.3527811. URL https://doi.org/10.1145/3524613.3527811.

Rachel Searcey. A coding translation to increase the efficiency of programmatic data analyses. 2023. doi: 10.1109/ccwc57344.2023.10099092.

Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. A survey on machine learning techniques for source code analysis. *CoRR*, abs/2110.09610, 2021. URL https://arxiv.org/abs/2110.09610.

NamJiho Shin and Jaechang. A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17(3):537–555, 6 2021.

Martin L. Shooman. Probabilistic models for software reliability prediction. In Walter Freiberger, editor, *Statistical Computer Performance Evaluation*, pages 485–502. Academic Press, 1972. ISBN 978-0-12-266950-7. doi: https://doi.org/10.1016/B978-0-12-266950-7.50029-3. URL https://www.sciencedirect.com/science/article/pii/B9780122669507500293.

Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory: Volume I Languages and Parsing*, volume 15. Springer Science & Business Media, 2012.

Shan Suthaharan. *Decision Tree Learning*, pages 237–269. Springer US, Boston, MA, 2016. ISBN 978-1-4899-7641-3. doi: 10.1007/978-1-4899-7641-3_10. URL https://doi.org/10.1007/978-1-4899-7641-3_10.

Maxim Tabachnyk and Stoyan Nikolov. Ml-enhanced code completion improves developer productivity. https://research.google/blog/ml-enhanced-code-completion-improves-developer-productivity/?utm_source=chatgpt.com, 2022. Accessed: 2025-02-27.

Sanket Tavarageri, Gagandeep Goyal, Sasikanth Avancha, Bharat Kaul, and Ramakrishna Upadrasta. Ai powered compiler techniques for dl code optimization. *arXiv: Programming Languages*, 2021.

P Thomas Schoenemann. Syntax as an emergent characteristic of the evolution of semantic complexity. *Minds and Machines*, 9:309–346, 1999.

Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. 02 2022. doi: 10.1145/3510003.3510067.

Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In Hugo Jair Escalante and Katja Hofmann, editors, *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, volume 133 of *Proceedings of Machine Learning Research*, pages 3–26. PMLR, 06–12 Dec 2021. URL https://proceedings.mlr.press/v133/turner21a.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. Enriching query semantics for code search with reinforcement learning. *Neural Networks*, 145:22–32, 2022a. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2021.09.025. URL https://www.sciencedirect.com/science/article/pii/S0893608021003877.

Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. Automating reinforcement learning architecture design for code optimization. In *Proceedings* *of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 129–143, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450391832. doi: 10.1145/3497776.3517769. URL https://doi.org/10.1145/3497776.3517769.

Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. Refactoring pattern matching. *Science of Computer Programming*, 78(11):2216–2242, 2013. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2012.07.014. URL https://www.sciencedirect.com/science/article/pii/S0167642312001426. Special section on Mathematics of Program Construction (MPC 2010) and Special section on methodological development of interactive systems from Interaccion 2011.

Shanshan Wang and Xiaohui Wang. Cross-language translation and comparative study of english literature using machine translation algorithms. *Journal of Electrical Systems*, 2024. doi: 10.52783/jes.3136.

Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. Cocosum: Contextual code summarization with multi-relational graph neural network, 2021. URL https://arxiv.org/abs/2107.01933.

Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

Min Xiao and Yuhong Guo. A novel two-step method for cross language representation learning. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/0ff39bbbf981ac0151d340c9aa40e63e-Paper.pdf.

Yan Xiao, Xinyue Zuo, Lei Xue, Kailong Wang, Jin Song Dong, and Ivan Beschastnikh. Empirical study on transformer-based techniques for software engineering, 2023. URL https://arxiv.org/abs/2310.00399.

Kaiyuan Yang, Junfeng Wang, and Zihua Song. Learning a holistic and comprehensive code representation for code summarization. *Journal of Systems and Software*, 203:111746, 2023. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2023.111746. URL https://www.sciencedirect.com/science/article/pii/S0164121223001413.