UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
DEPARTAMENTO DE ESTATÍSTICA E INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA APLICADA

IZAVAN DOS SANTOS CORREIA

IDENTIFYING PARALLELIZATION POINTS IN SEQUENTIAL CODES USING ARTIFICIAL INTELLIGENCE

Recife
2026

IZAVAN DOS SANTOS CORREIA

IDENTIFYING PARALLELIZATION POINTS IN SEQUENTIAL CODES USING ARTIFICIAL INTELLIGENCE

Dissertation presented to the *Programa de Pós-Graduação em Informática Aplicada* of the *Universidade Federal Rural de Pernambuco*, as a partial requirement for the degree of Master in Applied Informatics. Area of concentration: As indicated in the defense minutes.

Orientador: Prof. Dr. Tiago Alessandro Espínola Ferreira

Coorientador: Prof. Dr. Henrique Correia Torres Santos

Recife

2026

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Bibliotecário(a): Auxiliadora Cunha – CRB-4 1134

C824i Correia, Izavan dos Santos.
Identifying Parallelization Points in Sequential Codes Using Artificial Intelligence / Izavan dos Santos Correia. – Recife, 2026.
85 f.; il.

Orientador(a): Tiago Alessandro Espínola Ferreira.
Co-orientador(a): Henrique Correia Torres Santos.

Dissertação (Mestrado) – Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação em Informática Aplicada, Recife, BR-PE, 2026.

Inclui referências.

1. Paralelização automática. 2. Aprendizado computacional. 3. Modelos de transformadores. 4. análise de código - Fonte 5. Computação de alto desempenho. I. Ferreira, Tiago Alessandro Espínola, orient. II. Santos, Henrique Correia Torres, coorient. III. Título

CDD 004

IZAVAN DOS SANTOS CORREIA

IDENTIFYING PARALLELIZATION POINTS IN SEQUENTIAL CODES USING ARTIFICIAL INTELLIGENCE

Dissertation presented to the *Programa de Pós-Graduação em Informática Aplicada* of the *Universidade Federal Rural de Pernambuco*, as a partial requirement for the degree of Master in Applied Informatics.

Approved in: 25 / 02 / 2026.

EXAMINING BOARD

_____

Prof. Dr. Tiago Alessandro Espínola Ferreira (Orientador)
Universidade Federal Rural de Pernambuco

_____

Prof. Dr. Victor Wanderley Costa de Medeiros (Examinador Interno)
Universidade Federal de Pernambuco

_____

Prof. Dr. Rômulo César Carvalho de Araújo (Examinador Externo)
Instituto Federal de Pernambuco

To my family, for their unconditional support throughout my academic journey, and to my professors, whose guidance and encouragement were essential to the completion of this work.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Professor Tiago, for his guidance, support, and valuable contributions throughout the development of this work. I also thank my co-advisor, Professor Henrique, for his important insights and assistance during this research.

I am also grateful to Professor Rômulo for his guidance, support, and encouragement during the early stages of my academic journey, which played an important role in shaping my academic development.

I would like to thank the Universidade Federal Rural de Pernambuco, the Programa de Pós-Graduação em Informática Aplicada, and the IFROG Research Group for providing an inspiring academic environment and the necessary support for the completion of this work.

I am deeply thankful to my family and my girlfriend for their unconditional support, encouragement, and understanding throughout my academic journey.

Finally, I acknowledge the financial support provided by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), which granted the master's scholarship that made this research possible.

ABSTRACT

This dissertation investigates the use of Artificial Intelligence to automate the identification of parallelization opportunities in sequential programs. The research is motivated by limitations observed in the Parallel Experience for Sequential Code (PESC) framework, which enables distributed execution through efficient resource allocation but still relies on manual inspection to determine where parallelism can be safely introduced. This dependency represents a critical bottleneck in scalability and usability. To address this challenge, the thesis explores learning-based approaches capable of inferring dependency patterns directly from source code. The investigation is organized as a collection of two complementary studies that progressively increase representational complexity and realism. The first study evaluates lightweight deep learning architectures, including Deep Neural Networks and Convolutional Neural Networks, using token-based representations and synthetically generated programs with explicit dependency constraints. In the best experimental configuration, the CNN achieved 97.67% test accuracy, with F1-scores of 0.98 for dependent loops and 0.97 for independent ones, demonstrating that even simplified models can effectively recognize structural cues associated with parallelizability. The second study advances the approach by employing the Transformer-based model DistilBERT and by combining synthetic data with manually annotated real-world code. Despite the higher variability, the method reached a mean test accuracy of 99.60%, while maintaining an extremely low false positive rate (mean 0.0029), a crucial property for safe adoption in practical environments. Together, the results provide consistent evidence that Artificial Intelligence techniques can significantly reduce the need for manual analysis in distributed execution workflows. The dissertation contributes empirical validation, reproducible dataset construction strategies, and a pathway toward integrating intelligent classifiers into infrastructures such as PESC, promoting greater automation in high-performance computing contexts.

Keywords: automatic parallelization; deep learning; Transformers models; source code analysis; high-performance computing.

RESUMO

Esta dissertação investiga o uso de Inteligência Artificial para automatizar a identificação de oportunidades de paralelização em programas sequenciais. A pesquisa é motivada por limitações observadas na ferramenta *Parallel Experience for Sequential Code* (PESC), que viabiliza a execução distribuída por meio da alocação eficiente de recursos, mas ainda depende de inspeção manual para determinar onde o paralelismo pode ser introduzido com segurança. Essa dependência representa um gargalo crítico para a escalabilidade e a usabilidade. Para enfrentar esse desafio, a tese explora abordagens baseadas em aprendizado capazes de inferir padrões de dependência diretamente a partir do código-fonte. A investigação é organizada como uma coletânea de dois estudos complementares que aumentam progressivamente a complexidade representacional e o realismo dos experimentos. O primeiro estudo avalia arquiteturas leves de *deep learning*, incluindo Redes Neurais Densas e Redes Neurais Convolucionais, utilizando representações baseadas em tokens e programas sintéticos com restrições explícitas de dependência. Na melhor configuração experimental, a CNN alcançou 97,67% de acurácia em teste, com F1-scores de 0,98 para loops dependentes e 0,97 para loops independentes, demonstrando que mesmo modelos simplificados conseguem reconhecer pistas estruturais associadas à paralelizabilidade. O segundo estudo avança a proposta ao empregar o modelo baseado em Transformers DistilBERT e ao combinar dados sintéticos com códigos reais anotados manualmente. Apesar da maior variabilidade, o método atingiu acurácia média de 99,60% em teste, mantendo uma taxa de falso positivo extremamente baixa (média de 0,0029), característica essencial para adoção segura em ambientes práticos. Em conjunto, os resultados fornecem evidências consistentes de que técnicas de Inteligência Artificial podem reduzir significativamente a necessidade de análise manual em fluxos de execução distribuída. A dissertação contribui com validação empírica, estratégias reprodutíveis de construção e rotulagem de dados, e um caminho para integrar classificadores inteligentes a infraestruturas como o PESC, promovendo maior automação em contextos de computação de alto desempenho.

Palavras-chave: paralelização automática; aprendizado profundo; modelos Transformers; análise de código-fonte; computação de alto desempenho.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

SUMMARY

1 INTRODUCTION

In recent years, the continuous growth in computational demand has intensified the need for strategies capable of improving software performance and scalability. Parallel computing has become one of the primary mechanisms for exploiting modern multi-core and distributed infrastructures, enabling applications to execute faster and use available resources more efficiently (Santos et al., 2025). Nevertheless, a significant portion of existing software was originally developed following sequential paradigms, which makes the migration to parallel environments a complex and often manual endeavor.

Within this context, the Parallel Experience for Sequential Code (PESC) framework was introduced to enable the execution of sequential applications in distributed environments through intelligent workload distribution and the use of idle computational resources (Santos et al., 2025). The platform provides mechanisms to simulate user executions and allocate tasks across available machines, reducing execution time while minimizing the need for deep user intervention. Despite these advances, an important limitation persists: the identification of code regions that can be safely parallelized. In practice, this step has traditionally required manual inspection of the source code and explicit adaptations, such as the insertion of control structures associated with process ranks. This procedure is labor-intensive, difficult to scale, and subject to human error.

The research community has long investigated automatic strategies for detecting parallelism in sequential programs. Traditional approaches are predominantly rooted in compiler theory and rely on static dependence analysis, control-flow modeling, and polyhedral frameworks to determine whether loop iterations can be executed concurrently. Techniques implemented in systems such as PLuTo, ISL, and the Omega Calculator provide mathematically rigorous mechanisms for reasoning about loop-carried dependencies in affine programs (Pouchet et al., 2009; Zinenko et al., 2018). Although these methods offer strong formal guarantees, their applicability is often restricted in the presence of irregular memory accesses, dynamic behaviors, or complex indexing patterns commonly observed in real-world software.

In parallel with compiler-driven solutions, recent years have witnessed a rapid expansion in the adoption of Machine Learning techniques for source code analysis.

Early evidence demonstrated that statistical models operating on tokens or engineered features could capture relevant program properties (Allamanis et al., 2016), and broader surveys consolidated the role of deep learning across numerous software engineering tasks (Allamanis et al., 2018; White et al., 2016). The emergence of Transformer architectures further advanced representation learning by enabling the modeling of long-range dependencies within code (Vaswani et al., 2017). Pre-trained models such as CodeBERT and GraphCodeBERT achieved state-of-the-art performance in diverse applications (Feng et al., 2020; Guo et al., 2020), while more specialized efforts began targeting parallelization itself, including approaches such as PragFormer (Chen et al., 2024). Prior investigations that motivated the developments presented in this dissertation demonstrated that even lightweight neural models trained on token sequences can successfully discriminate loop independence (Correia et al., 2025). Together, these results suggest that learning-based systems constitute a promising avenue for supporting automatic identification of parallelization opportunities, motivating the studies that compose this thesis.

This dissertation investigates the hypothesis that Artificial Intelligence, particularly deep learning and Transformer-based language models, can automate the detection of potential parallelization points in sequential programs. Instead of relying exclusively on formal compiler analyses, we explore whether learning-based systems can infer dependency patterns directly from code representations. If successful, such models may substantially reduce the manual effort required in the PESC workflow and broaden the practical applicability of distributed execution.

1.1 OBJECTIVES

The general objective of this dissertation is to develop and evaluate intelligent models capable of automatically identifying loop-level parallelization opportunities in sequential source code, contributing to the reduction of manual intervention in distributed execution environments.

To achieve this goal, the following specific objectives are defined:

- To investigate whether lightweight neural architectures can recognize dependency patterns under controlled experimental conditions;
- To design and generate labeled datasets that enable systematic evaluation of loop independence;

- To assess the predictive stability and robustness of different deep learning models;
- To explore Transformer-based representations capable of extracting contextual information directly from raw code;

1.2 DOCUMENT STRUCTURE

The research is structured as a collection of two complementary scientific articles, each addressing a different level of representational and architectural complexity while pursuing the same overarching objective: the automatic identification of loop-level parallelization opportunities.

The **first article**, titled *"Discovering Software Parallelization Points Using Deep Neural Networks,"* investigates whether lightweight neural architectures are sufficient to capture signals of loop independence under controlled conditions. Due to the scarcity of labeled datasets, the study employs a genetic algorithm to synthesize programs with explicit dependency constraints, producing balanced classes of independent and dependent loops. Using exclusively token-based representations, Deep Neural Networks (DNN) and Convolutional Neural Networks (CNN) are evaluated through repeated executions in order to analyze predictive stability and robustness. The results demonstrate that even simplified representations contain meaningful information for distinguishing parallelizable structures, establishing an important proof of concept and clarifying the lower bounds of representational sufficiency for this task.

The **second article**, titled *"Automatic Identification of Parallelizable Loops Using Transformer-Based Source Code Representations,"* expands the scope toward more realistic scenarios. In this stage, we adopt DistilBERT, a computationally efficient Transformer model capable of producing contextual embeddings directly from raw source code. The experimental dataset combines large volumes of synthetic samples with manually annotated real-world programs, increasing structural variability and approximating practical development conditions. The findings indicate that the Transformer-based approach preserves high accuracy while simplifying preprocessing requirements and achieving extremely low false positive rates, an essential characteristic in environments where incorrect parallelization decisions may compromise execution safety. Specifically, a false positive, that is, classifying a

dependent loop as parallelizable, may lead compilers or runtime systems to apply parallel execution strategies to inherently sequential code, introducing race conditions, data corruption, or non-deterministic behavior that are difficult to diagnose and reproduce.

Taken together, the two studies establish a progression from controlled experimentation with minimal representations to broader applicability with contextual language models. From a scientific perspective, this dissertation contributes empirical evidence that automatic parallelization support can be effectively assisted by learning-based techniques, demonstrates viable alternatives with different computational costs, and provides reproducible methodologies for dataset construction and evaluation. From a practical standpoint, the results reinforce the feasibility of incorporating intelligent classifiers into infrastructures such as PESC, reducing dependence on manual code inspection and moving toward scalable automation.

The remainder of this dissertation is organized as follows. After this introductory chapter, the next sections present the two articles in their complete form. The document concludes with general considerations that integrate the findings, discuss limitations, and outline promising directions for future research.

2 ARTICLE 1 – DISCOVERING SOFTWARE PARALLELIZATION POINTS USING DEEP NEURAL NETWORKS

2.1 ABSTRACT

This study investigates whether lightweight deep learning models can distinguish parallelizable loops using only token-based representations of source code under controlled experimental conditions. To address the scarcity of labeled data for loop parallelizability analysis, we employ a genetic algorithm to generate synthetic Python loops belonging to two simplified categories: independent loops without data dependencies and dependent loops whose dependency structure prevents parallel execution. The generated code is tokenized and preprocessed to construct a dataset of approximately 4,000 samples. Two neural architectures—a Deep Neural Network (DNN) and a Convolutional Neural Network (CNN)—are evaluated on this task, with performance stability assessed across 30 independent training runs. While the CNN achieves marginally higher mean accuracy, both models exhibit comparable variability. Additional experiments with reduced datasets highlight the influence of dataset diversity on classification performance. Rather than proposing a new parallelization technique, this work provides empirical evidence on the minimum representational and architectural complexity required to capture loop-level parallelizability signals from lexical code features, thereby clarifying the potential and limitations of token-based learning approaches in this domain.

Keywords: Loop Classification; Parallelization Detection; Deep Neural Networks; Convolutional Neural Networks; Genetic Algorithm; Code Tokenization.

2.2 INTRODUCTION

In today's rapidly evolving technological landscape, optimizing software performance has become increasingly crucial. Parallel programming stands out as one of the most effective strategies to improve execution speed and resource utilization, allowing applications to fully exploit modern multi-core and heterogeneous computing architectures (Santos et al., 2025).

As parallel computing becomes more prevalent, the identification of parallelization opportunities has gained renewed importance. Loops are among the most promising targets for optimization, since their iterations can often be executed concurrently when no loop-carried dependencies are present. However, automatically determining whether a loop can be safely parallelized remains a challenging task, primarily due to the difficulty of detecting subtle data dependencies that may prevent parallel execution (Zinenko et al., 2018; Pouchet et al., 2009; Omega Calculator approach).

Traditional solutions to this problem are largely based on static program analysis. Techniques such as dependence analysis and polyhedral frameworks can formally prove loop independence, but they often struggle when faced with irregular control flow, non-affine memory accesses, or complex code transformations (Zinenko et al., 2018; Pouchet et al., 2009; Omega Calculator approach). More recent approaches have explored reinforcement learning strategies to learn parallelization policies automatically; however, these methods typically require extensive training and deep integration with sophisticated compiler infrastructures, making them computationally expensive and difficult to deploy in lightweight analysis scenarios.

In parallel, machine learning (ML) approaches—particularly those based on deep learning—have emerged as a promising alternative for program analysis tasks. Models built upon Graph Neural Networks (GNNs) and Transformer architectures have demonstrated strong performance by leveraging rich code representations such as Abstract Syntax Trees (ASTs), control-flow graphs, and other structural program features (Shen et al., 2021; Chen et al., 2022; Harel et al., 2025; Mahmud et al., 2025). Foundational surveys in this area emphasize the growing relevance of deep learning for source code analysis, highlighting its ability to capture both structural and semantic patterns in programs (Allamanis et al., 2018; Sharma, 2021; Liu et al., 2019). While these approaches demonstrate strong results, they implicitly assume that rich

structural representations and highly expressive models are necessary to identify loop-level parallelization opportunities. However, it remains unclear whether such representational and architectural complexity is strictly required for detecting fundamental loop-carried dependencies, particularly under controlled conditions.

Despite their effectiveness, many existing approaches rely on complex representations, large model architectures, and substantial computational resources. In contrast, this work deliberately reframes loop parallelizability detection as a representational sufficiency problem rather than a competition for maximal predictive performance. Specifically, we investigate whether purely token-based representations of source code, when processed by lightweight neural networks, are sufficient to capture the minimal signals required to distinguish independent from dependent loops under controlled experimental conditions.

A key obstacle in supervised learning approaches for parallelism detection is the lack of publicly available datasets containing loops explicitly labeled according to their dependency structure. To address this limitation, we adopt a data generation strategy based on a genetic algorithm that automatically synthesizes Python loops with controlled dependency patterns. Rather than aiming to fully replicate real-world software complexity, this synthetic dataset is designed to enable precise supervision and controlled experimentation, allowing us to isolate the relationship between token-level representations, model complexity, and loop dependence signals.

Using this dataset, we evaluate two lightweight neural architectures—a Deep Neural Network (DNN) and a Convolutional Neural Network (CNN) trained exclusively on tokenized representations of source code. The models are assessed through multiple independent executions to account for stochastic effects during training. In addition, we investigate the application of Principal Component Analysis (PCA) for dimensionality reduction, analyzing its impact on model performance and generalization under reduced feature representations. Rather than proposing a new parallelization algorithm or compiler optimization technique, the contributions of this work lie in empirically characterizing the lower bounds of representational and architectural complexity required for loop-level parallelizability classification using deep learning.

The main contributions of this paper can be summarized as follows:

- A reproducible method for generating labeled loop-level parallelization data using a genetic algorithm that synthesizes independent and dependent Python loops.
- An empirical evaluation of lightweight, token-based neural models (DNN and CNN) for classifying loop parallelizability without relying on complex program representations.
- A stability-oriented experimental analysis based on multiple independent runs, providing insights into performance variability and robustness.
- An investigation of the effect of dimensionality reduction via PCA on classification performance and representational efficiency.

The results suggest that deep learning models can capture meaningful signals related to loop independence directly from tokenized code, offering a cost-effective and rapid alternative to more complex analysis techniques within controlled experimental settings.

The remainder of this paper is organized as follows. Section 2.3 reviews related work on loop parallelization and ML-based code analysis. Section 2.4 describes the proposed methodology. Section 2.5 presents the experimental results, followed by a discussion in Section 2.6. Finally, Section 2.7 concludes the paper and outlines directions for future research.

2.3 RELATED WORK

The identification of parallelization opportunities, particularly at the loop level, has long been a central problem in compiler optimization and program analysis. Classical approaches are predominantly based on static analysis techniques, including data-flow, control-flow, and dependence analysis, which aim to determine whether loop iterations are independent and can therefore be executed in parallel. Among these, polyhedral frameworks such as PLuTo, ISL, and the Omega Calculator provide precise mathematical models for reasoning about loop-carried dependencies in affine programs (Zinenko et al., 2018; Pouchet et al., 2009; Omega Calculator approach). While highly effective for structured code with regular access patterns, these techniques often struggle when applied to real-world programs that exhibit irregular control flow, non-affine memory accesses, or complex indexing behavior.

Unlike these classical compiler-based techniques, which aim at formally proving the absence of dependencies under strict structural assumptions, the present work does not attempt to replace or approximate static dependence analysis. Instead, it addresses a complementary question: whether minimal lexical signals present in source code tokens are sufficient for a learning-based model to distinguish independent from dependent loops under controlled conditions. This distinction in scope is fundamental, as our objective is representational analysis rather than exact dependence verification.

In recent years, machine learning (ML) techniques have been increasingly explored as a complementary or alternative approach to traditional compiler analyses. Surveys such as (Allamanis et al., 2018) highlight the growing application of deep learning models to a wide range of program analysis and software engineering tasks, demonstrating their ability to capture both syntactic and semantic code patterns. This trend has also influenced research on automatic parallelization, where ML models are trained to infer parallelizability from program representations rather than relying solely on handcrafted rules.

Several studies have proposed learning-based methods that leverage rich structural representations of code. For instance, Shen et al. (2021) introduced a Graph Neural Network (GNN) approach that analyzes contextual flow graphs using graph convolutional networks to detect parallelizable regions. Similarly, Chen et al. (2022) proposed a multi-view learning framework that integrates dependency information into GNN-based models to improve loop parallelization decisions. These approaches demonstrate strong performance by explicitly modeling program structure, but they typically depend on complex intermediate representations and incur substantial computational overhead during preprocessing and training.

More recent work has extended these ideas through the use of Transformer architectures and Large Language Models (LLMs). Mahmud et al. (2025) presented AutoParLLM, a framework that combines GNN-based representations with LLMs to discover parallelism and automatically generate parallel code, introducing the *OMPScore* metric to evaluate the quality of the generated output. In a related direction, Harel et al. (2025) proposed PragFormer, a Transformer-based model designed to classify for-loops that benefit from OpenMP directives, including the prediction of data-sharing clauses. While these models achieve impressive results, they typically require

large training corpora, sophisticated representations, and significant computational resources.

In contrast to these approaches, which rely on rich structural representations such as abstract syntax trees, control-flow graphs, or intermediate compiler representations, our study deliberately restricts the input to raw token sequences. While GNN- and Transformer-based models aim to maximize predictive accuracy by exploiting detailed program structure, our work investigates a different research question: the minimum representational and architectural complexity required to capture fundamental loop-carried dependency signals. Rather than aiming to match or replace compiler-grade analyses or state-of-the-art graph-based models, we focus on assessing whether loop independence can be inferred from token-level patterns alone. By training lightweight Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) on synthetically generated loops produced via a genetic algorithm, we position our study as an exploration of the lower bounds of representational sufficiency for loop-level parallelizability classification under controlled experimental conditions.

Direct experimental comparison with existing GNN or Transformer-based methods is beyond the scope of this study, as such models operate on fundamentally different input representations and problem formulations. Comparing performance metrics across approaches that rely on distinct levels of semantic and structural information would conflate representational power with architectural complexity. Instead, our work focuses on isolating the contribution of token-level information, offering a controlled analysis that complements, rather than competes with, existing state-of-the-art methods.

This positioning distinguishes our work from prior studies by emphasizing simplicity, computational efficiency, and controlled experimentation. At the same time, it complements existing research by providing empirical evidence on what can be achieved without relying on complex structural representations, thereby helping to clarify the trade-offs between model expressiveness, resource requirements, and practical applicability in parallelism detection tasks.

2.4 METHODOLOGY

This study follows a structured, multi-stage methodology designed to assess whether lightweight deep learning models can identify loop-level parallelization

opportunities using exclusively token-based representations of source code. The workflow comprises four main stages: (i) synthetic data generation using a genetic algorithm, (ii) tokenization and numerical encoding of code samples, (iii) optional dimensionality reduction via Principal Component Analysis (PCA), and (iv) training and evaluation of neural models under multiple independent executions.

2.4.1 Programming Code Generation

A genetic algorithm was employed to generate labeled code samples due to the absence of publicly available benchmarks containing loops annotated by dependence structure. Real-world repositories rarely include explicit information indicating whether a loop is parallelizable, making supervised training impractical without extensive manual annotation. Genetic algorithms, originally formalized by Holland as adaptive search mechanisms (Holland, 1975), enable controlled synthesis of diverse loop structures by varying operations, variable usage, data access patterns, and dependency relationships. This approach ensures the production of clear independent and dependent loop cases that challenge dependency analysis and support meaningful learning.

The code generator was implemented using the DEAP Python library[1]. Python served as both the language for the generator and for the produced code samples, although the methodology is generalizable to other languages.

Two separate generators were built: one for parallelizable loops (positive class) and one for non-parallelizable loops (negative class). Both share the same underlying GA structure, differing only in loop construction rules. Independent loops avoid reusing loop variables inside the loop body, whereas dependent loops intentionally reuse variables or introduce loop-carried dependencies.

Generated samples include typical program components such as functions, variables, operations, conditionals, library imports, and loops. The loop structure determines the assigned class.

Each individual in the GA population encodes a complete Python program as a string. The initial population is randomly generated and evolves through crossover and mutation. Crossover exchanges code line segments between individuals, while mutation applies edits such as variable replacements or insertion of statements.

[1]DEAP documentation: https://deap.readthedocs.io

The fitness function rewards structural attributes such as number of functions, loops, variables, and successful compilation. Invalid or trivial programs are penalized.

The fitness score *f* for an individual program *P* is defined as:

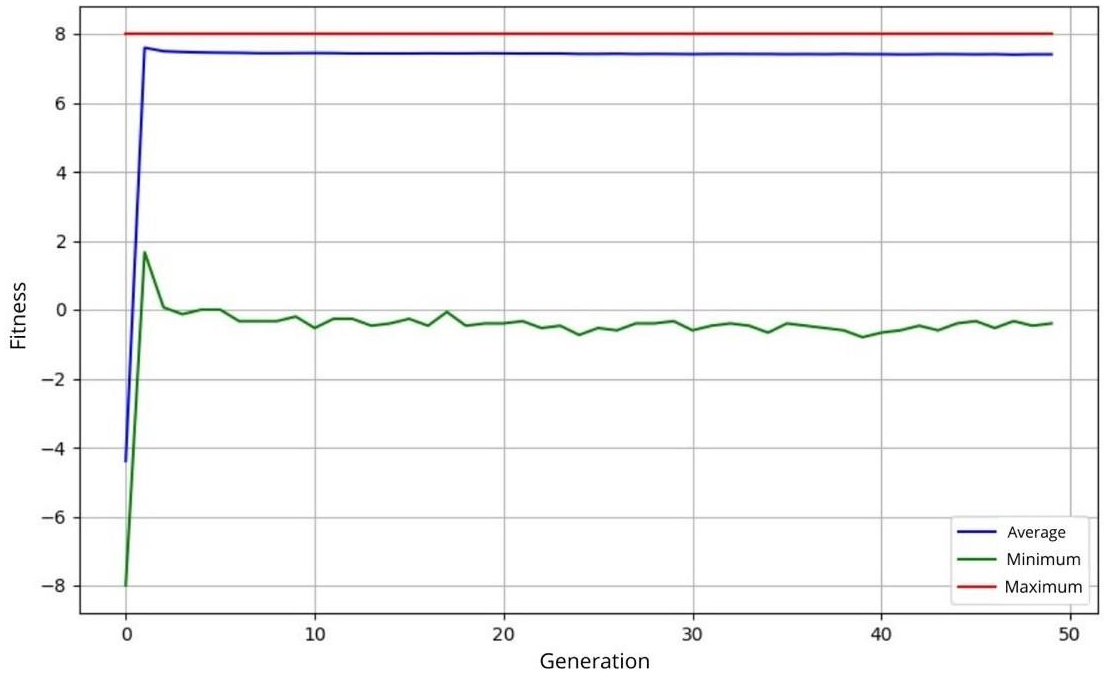
$$f(p) = \sum_{i=1}^{7} s_i \; + \; s_{comp} \quad (1)$$

Where each s_i corresponds to a structural feature:

- s_1: number of import statements between 1–12 ⇒ +1, otherwise −1;
- s_2: number of def functions between 2–8 ⇒ +1, otherwise −1;
- s_3: number of if statements between 2–8 ⇒ +1, otherwise −1;
- s_4: number of print calls between 2–8 ⇒ +1, otherwise −1;
- s_5: number of variables between 2–100 ⇒ +1, otherwise −1;
- s_6: number of for loops between 1–6 ⇒ +1, otherwise −5;
- s_7: number of lines between 10–150 ⇒ +1, otherwise −1;
- s_{comp}: successful compilation ⇒ +1, otherwise −5.

The evolution process uses a population of 10,000 individuals for 50 generations, with roulette selection, crossover probability 0.9, and mutation probability 0.1.

Figure 1 – shows the fitness evolution across 30 runs of the genetic algorithm



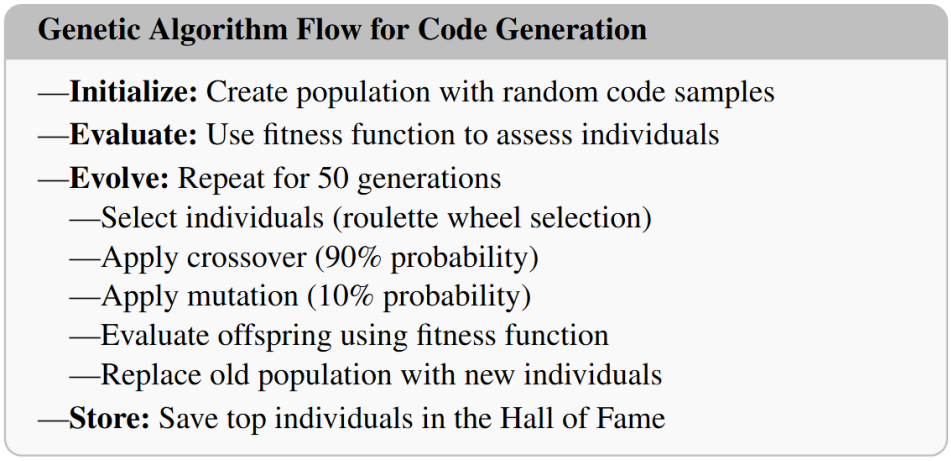
Source: Prepared by the authors, 2026.

Figure 1 illustrates the optimization progress across 30 independent runs. Fitness improves significantly in early generations and stabilizes toward the end. For the purposes of this study, 50 generations were adequate to generate structurally diverse programs aligned with the fitness criteria.

A Hall of Fame mechanism from DEAP was used to retain the top 500 individuals out of 10,000 candidates. All selected samples were validated in VSCode[2] and compiled without errors, confirming the robustness of the evolutionary process.

A high-level overview of the entire evolutionary pipeline—including initialization, selection, crossover, mutation, and elitism—is presented in Figure 2.

Figure 2 – Evolutionary process for generating labeled code samples



Source: Prepared by the authors, 2026.

The genetic algorithm employed in this work is not proposed as a novel optimization technique. Instead, it serves as a practical and systematic mechanism for synthesizing labeled loop instances under explicit dependency constraints. Its role is to enable scalable data generation with controlled properties in the absence of publicly available annotated benchmarks, rather than to introduce algorithmic innovation in evolutionary computation. To illustrate the type of code produced by the genetic algorithm, Listings 1 and 2 present simplified examples of the two classes generated during dataset construction. Listing 1 shows a dependent loop, in which the iteration bound is modified within the loop body, creating a loop-carried dependency that prevents safe parallelization. Listing 2 shows an independent loop, in which each

[2] VSCode: https://code.visualstudio.com

iteration computes its result exclusively from the loop control variable, with no data dependency between iterations.

Listing 1 – Example of a synthesized dependent loop

```
failure = 14
for produtiva in range(failure):
    failure = produtiva # loop-carried dependency: modifies iteration bound
```

Source: Prepared by the authors, 2026.

Listing 2 – Example of a synthesized independent loop

```
equaciona = 28
for destilasse in range(equaciona):
    descoberto = destilasse * 256 # no dependency between iterations
```

Source: Prepared by the authors, 2026.

2.4.2 Loop Classification Criteria

Each generated loop is automatically labeled as *independent* or *dependent*. A loop is considered **parallelizable** when all variables in its body are locally defined or iteration-invariant, ensuring that no iteration depends on values produced by others. Conversely, a loop is labeled **non-parallelizable** when it accesses external mutable state or relies on values that may change across iterations, indicating potential loop-carried dependencies.

In this study, loop dependence is defined at the semantic level rather than through compiler-grade data-flow analysis. Specifically, the generators enforce or prohibit loop-carried dependencies by controlling variable scope and reuse. Dependent loops include at least one variable whose value is modified in one iteration and subsequently read or modified in another iteration, such as accumulators, shared lists, or externally defined mutable objects. Independent loops, by contrast, operate exclusively on iteration-local variables or read-only data, ensuring that no iteration affects another. While this approach does not capture all possible dependence patterns, it provides a controlled and unambiguous labeling mechanism suitable for supervised learning and reproducible experimentation.

Within this framework, the classification task is deliberately restricted to a binary distinction between loops that exhibit explicit loop-carried dependencies and those that do not. More nuanced dependency categories, such as conditional dependencies, loop-independent dependencies, or dependencies arising from complex data-flow interactions, are outside the scope of the present study. This simplification is intentional and allows the investigation to focus on whether minimal token-based representations and lightweight neural architectures can detect fundamental dependency signals under controlled conditions. It is acknowledged that real-world codebases may contain a broader spectrum of dependency patterns, and that restricting the classification to a binary scheme represents a deliberate scoping decision rather than a claim of completeness. Establishing reliable detection of the most fundamental dependency class constitutes a necessary prerequisite before extending the framework to handle finer-grained dependency taxonomies, which remains a direction for future work.

2.4.3 Dataset Creation

A dataset of 4000 labeled samples was constructed, comprising two balanced classes:

- 2000 independent (parallelizable) loops;
- 2000 dependent (non-parallelizable) loops.

The independent-loop generator was executed four times to produce 2000 samples, and the dependent-loop generator was similarly executed four times, resulting in a balanced dataset suitable for supervised learning and controlled experimentation.

As the dataset is synthetically generated, it is important to acknowledge its limitations. The samples are intentionally constrained to isolate loop-carried dependencies under simplified conditions and therefore do not capture the full spectrum of dependency patterns found in real-world codebases, such as deeply nested loops, complex control flow, function calls, aliasing, or interactions with external libraries. Consequently, the dataset is intended as a first-step experimental benchmark that supports systematic analysis of representational and architectural requirements, rather than as a substitute for compiler-grade dependence analysis on production software.

At the same time, the synthetic loops produced by the proposed genetic algorithm reflect dependency motifs that are commonly observed in practical programs. In particular, dependent loops frequently implement accumulator-style updates, in which a variable is read and modified across iterations, creating explicit loop-carried dependencies analogous to those found in summations, reductions, and iterative state updates. Independent loops, by contrast, operate on iteration-local variables or read-only data, reflecting typical data-parallel patterns such as element-wise computations. Moreover, the generated programs incorporate realistic structural elements, including function definitions, conditional statements, multiple loops, and library importswhich increase lexical and contextual diversity at the token level. While this design does not aim to replicate the full semantic complexity of real-world software, it ensures that the dataset captures representative dependency patterns encountered in practice, supporting a meaningful and controlled investigation of token-based loop parallelizability detection.

2.4.4 Tokenization and Preprocessing

Each loop was tokenized using Python's tokenize library[3]. The tokenizer extracts lexical units such as keywords, identifiers, literals, and operators. Each unique token received an integer ID stored in a JSON dictionary that ensured consistent mapping across the dataset. Because loop lengths vary, token sequences also vary. Sequences were padded to the maximum length observed in the dataset, preserving token order. This integer-based representation aligns with the goal of assessing minimal representational complexity.

For illustrative purposes, the following example demonstrates the tokenization pipeline applied to a representative loop instance. Given the loop below:

Listing 3 – Representative loop instance used for tokenization illustration

```
for destilasse in range(equaciona):
     descoberto = destilasse * 256
```

Source: Prepared by the authors, 2026.

The tokenizer extracts the following lexical units:

Listing 4 – Token sequence extracted from the representative loop

```
['for', 'destilasse', 'in', 'range', '(', 'equaciona', ')', ':',
'descoberto', '=', 'destilasse', '*', '256']
```

Source: Prepared by the authors, 2026.

Each token is then mapped to a unique integer ID via the JSON dictionary, producing the following integer sequence:

Listing 5 – Integer ID sequence after dictionary mapping

```
[12, 45, 8, 67, 3, 89, 4, 11, 102, 5, 45, 7, 201]
```

Source: Prepared by the authors, 2026.

It is worth noting that token-based representations inevitably discard structural and semantic information that would be available in richer program representations, such as abstract syntax trees or data-flow graphs. For example, two loops may share similar token sequences while exhibiting different semantic behaviors due to variable aliasing or control-flow context. In this study, such information loss is accepted as part of the experimental design, as the primary objective is to evaluate the discriminative power of minimal lexical representations rather than to achieve complete semantic equivalence with compiler analyses.

2.4.5 Dimensionality Reduction with PCA

High-dimensional padded sequences may increase training time or lead to overfitting in lightweight models. To evaluate whether more compact representations still preserve essential discriminative signals, Principal Component Analysis (PCA) was applied as an optional dimensionality-reduction step.

The original feature representation consists of 380 dimensions, corresponding to the number of components retained by PCA when configured to preserve 100% of the variance of the integer ID sequences representing each loop instance — effectively capturing the full information content of the tokenized representation prior to any compression. Applying PCA with progressively lower variance thresholds significantly reduces the input dimensionality:

- **100% variance**: 380 components (no reduction);
- **95% variance**: reduced to 309 components;

- **90% variance**: reduced to 269 components;
- **85% variance**: reduced to 237 components;
- **80% variance**: reduced to 208 components.

These results show the effective compression achieved by PCA at each threshold, allowing us to analyze how much redundancy exists in the original representation and to assess the trade-off between dimensionality and predictive performance.

Because PCA does not preserve token ordering, it is not intended for deployment in a production pipeline. Instead, it is used here to support robustness analysis and to estimate the minimum representational complexity required for reliable model performance.

2.4.6 Data Processing and Preparation

Processed sequences were normalized and divided into training (70%), validation (15%), and testing (15%) sets. All PCA configurations used identical splits to ensure consistent comparisons.

2.4.7 Model Selection

The model architectures adopted in this study were defined after preliminary exploratory experiments and guided by the overarching objective of minimizing architectural complexity. Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) were selected due to their simplicity, low computational cost, and suitability for fixed-length numeric representations derived from tokenized source code. Rather than pursuing state-of-the-art performance, the goal of this work is to assess whether lightweight neural architectures are sufficient to distinguish parallelizable from non-parallelizable loops based solely on token-level information.

More expressive sequence models, such as recurrent neural networks or Transformer-based architectures, were deliberately excluded from the experimental design. While such models have demonstrated strong performance in prior work, their increased representational capacity and parameter count could obscure the relationship between input representation and model behavior, making it difficult to isolate the contribution of token-level information. By focusing on DNN and CNN models, the analysis prioritizes controlled experimentation and enables a clearer

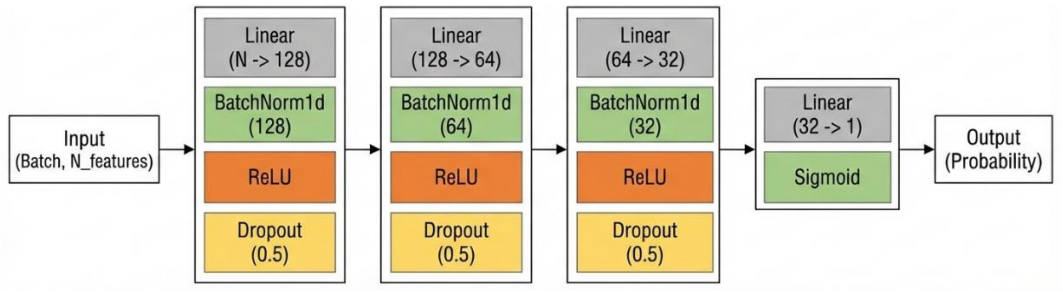
assessment of the minimum architectural complexity required for loop-level parallelizability classification.

2.4.8 DNN Model Architecture

A deep neural network was implemented using PyTorch (https://pytorch.org/docs/stable/index.html). After exploratory trials, the selected architecture (Figure 3) consisted of:

- Layer 1: 128 units, Batch Normalization, ReLU, Dropout (0.5);
- Layer 2: 64 units, Batch Normalization, ReLU, Dropout (0.5);
- Layer 3: 32 units, Batch Normalization, ReLU, Dropout (0.5);
- Output: 1 unit with sigmoid activation.

Figure 3 – Architecture of the proposed Deep Neural Network (DNN) model



Source: Prepared by the authors, 2026.

The model was trained for 1000 epochs using Binary Cross-Entropy Loss (BCELoss) (https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html) and the Adam optimizer (Kingma; Ba, 2014) with learning rate 0.001 and batch size 4. Training accuracy and loss were recorded each epoch. Final evaluation used accuracy, loss, confusion matrix, and classification report. A small batch size was intentionally adopted to increase gradient stochasticity and to better assess model robustness under limited data conditions.

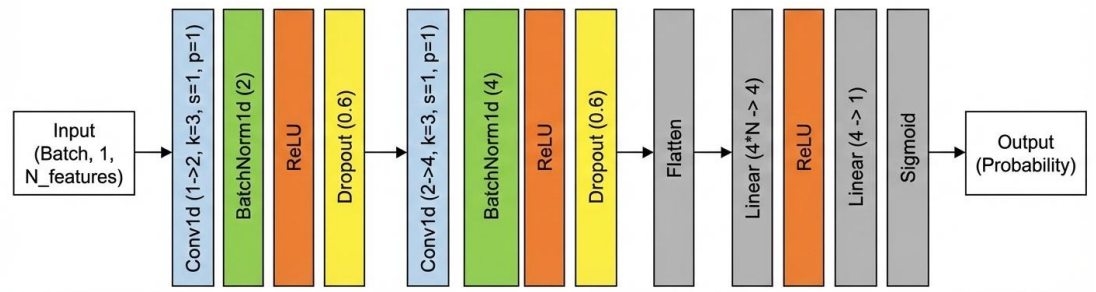
2.4.9 CNN Model Architecture

A convolutional neural network was also implemented in PyTorch. The selected architecture (Figure 4) contained:

- Convolution 1: 2 filters, kernel size 3, stride 1, padding 1; Batch Normalization, ReLU, Dropout (0.6);

- Convolution 2: 4 filters, kernel size 3, stride 1, padding 1; Batch Normalization, ReLU, Dropout (0.6);
- Fully connected layer: 4 units with ReLU;
- Output: 1 sigmoid unit.

Figure 4 – Architecture of the proposed Convolutional Neural Network (CNN) model



Source: Prepared by the authors, 2026.

The model was trained under the same settings used for the DNN: 1000 epochs, Adam optimizer, BCELoss, learning rate 0.001, and batch size 4. Evaluation metrics matched those used for the DNN.

2.4.10 Training and Evaluation

Both models were trained and evaluated across 30 independent runs to capture variability from random initialization and shuffling. Evaluation on the test set used accuracy, loss, confusion matrix, and classification metrics. Training history plots were generated to illustrate accuracy and loss progression across epochs.

Running each experiment 30 times follows common empirical practices in machine learning to reduce the influence of random initialization and stochastic training effects, providing more reliable performance estimates.

2.5 RESULTS

This section presents the experimental results for both DNN and CNN models, based on 30 executions each to ensure statistical reliability and account for random initialization and data shuffling effects. Evaluation metrics include accuracy, precision, recall, F1-score, and Binary Cross-Entropy Loss.

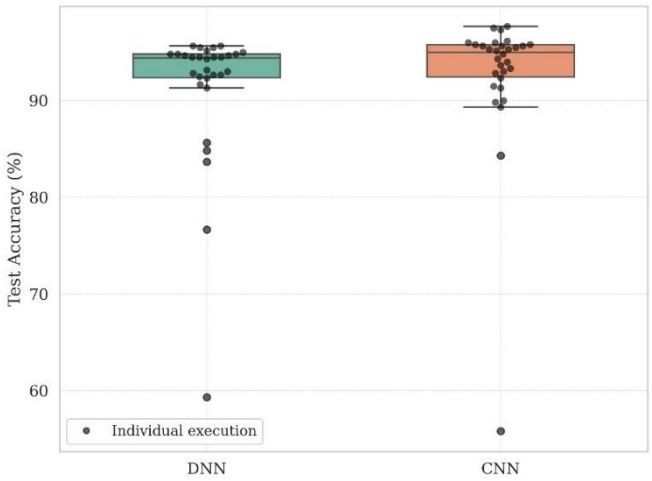
The analysis follows a structured approach: we first provide an overview of the 30 executions using boxplots to visualize performance distributions. We then examine

the average behavior of both models, followed by Principal Component Analysis (PCA) to assess performance across different training set proportions. Finally, we analyze the best and worst executions for each model to understand their variability and stability.

2.5.1 Performance Distribution Across 30 Executions

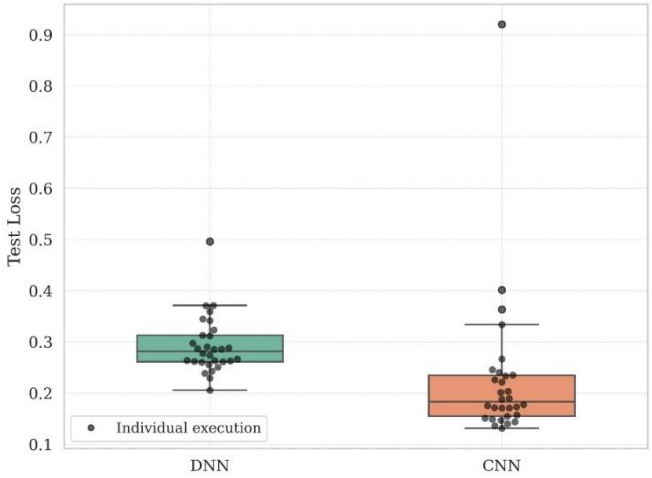
To evaluate model stability using the full dataset, each experiment was repeated 30 times with 100% of the original variance retained.

Figure 5 – Boxplots of test accuracy for DNN and CNN models using 100% of retained PCA variance across 30 executions



Source: Prepared by the authors, 2026.

Figure 6 – Boxplots of test loss for DNN and CNN models using 100% of retained PCA variance across 30 executions



Source: Prepared by the authors, 2026.

Figures 5 and 6 shows boxplots of test accuracy and test loss for both models, with individual points representing each execution. These plots illustrate performance distributions, including spread, central tendency, and outliers, highlighting the variability observed across independent executions. Additionally, the distributional behavior provides insights into the consistency of the optimization process: narrower boxes indicate more stable convergence, while wider spreads suggest sensitivity to initialization or training dynamics. These patterns characterize the stability of each architecture under identical experimental conditions and reflect performance variability across independent executions.

A Kolmogorov–Smirnov (KS) test was conducted to statistically compare the **test accuracy distributions** of both models at a 95% confidence level (AN, 1933; SMIRNOV, 1948). The test yielded a KS statistic of 0.3333 and a *p*-value of 0.0708, indicating no significant difference between the accuracy distributions.

However, when applied to the **test loss distributions** at the same confidence level, the KS test resulted in a statistic of 0.7000 and a *p*-value below 0.0001, demonstrating a statistical difference. Although the CNN exhibited lower error values, the results indicate no statistically significant difference in classification accuracy between the two models, motivating further analysis of their behavior.

2.5.2 Average Performance Using the Full Dataset

We now analyze the average behavior of both models across the 30 independent runs using the full training dataset. The 95% confidence intervals were calculated using Student's *t*-test (STUDENT, 1908).

The DNN model achieved an average test accuracy of **91.37%** (standard deviation: **7.41**, median: **94.41%**, CI: **[88.59, 94.13]**). The CNN model showed slightly better performance with an average accuracy of **92.70%** (standard deviation: **7.53**, median: **95.00%**, CI: **[89.89, 95.51]**).

Both models exhibited considerable performance ranges across the 30 executions. The DNN achieved maximum and minimum accuracies of **95.67%** and **59.33%**, respectively, while the CNN ranged from **97.67%** to **55.83%**. These extremes reflect the variability observed across different training executions.

Tables 1 and 2 summarize these statistics, including average performance, variability measures, and confidence intervals.

Table 1 – Average, standard deviation, and median test accuracy across 30 runs using 100% of the data

Model	Mean (%)	Std (%)	Median (%)
DNN	91.37	7.41	94.41
CNN	92.37	7.53	95.00

Source: Prepared by the authors, 2026.

Table 2 – Best, worst, and 95% confidence interval of test accuracy for DNN and CNN

Model	Best (%)	Worst (%)	95% CI of the Mean
DNN	95.67	59.33	[88.59, 94.13]
CNN	97.67	55.83	[89.89, 95.51]

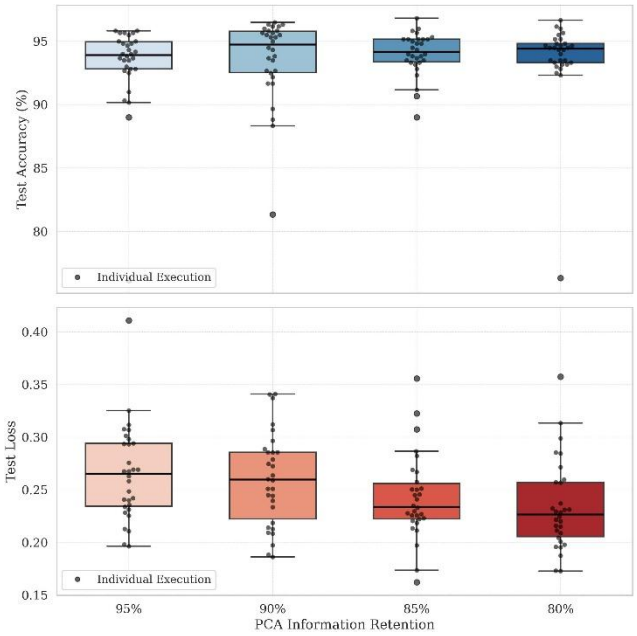
Source: Prepared by the authors, 2026.

2.5.3 Data Proportion Impact on Model Performance

We evaluated model performance under different levels of data compression by applying PCA while retaining 95%, 90%, 85%, and 80% of the original variance.

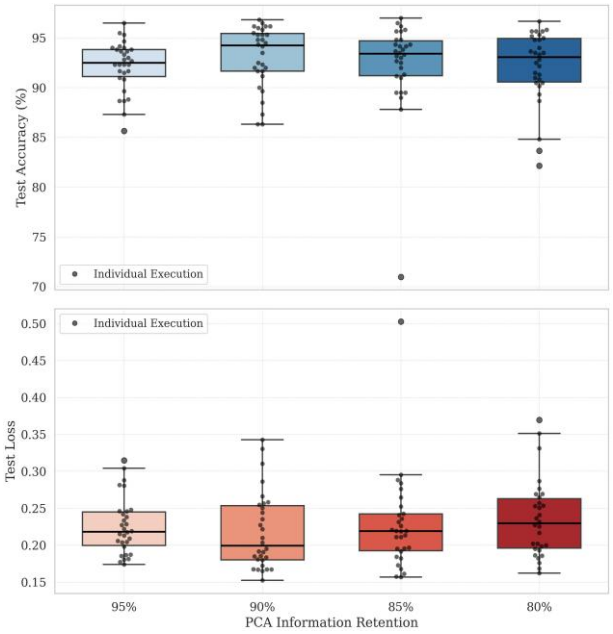
Figures 7 and 8 presents boxplots of test accuracy and test loss for the DNN (left) and CNN (right) models under different PCA variance retention levels across 30 executions.

Figure 7 – Test accuracy (top) and test loss (bottom) for the DNN model under different PCA variance retention levels across 30 executions



Source: Prepared by the authors, 2026.

Figure 8 – Test accuracy (top) and test loss (bottom) for the CNN model under different PCA variance retention levels across 30 executions



Source: Prepared by the authors, 2026.

For each proportion, boxplots of test accuracy and loss across 30 executions are presented in Figures 7 and 8, which combines the results for both the DNN and CNN models under all four variance retention levels for direct comparison.

Overall, both models demonstrate stable performance across all variance retention levels, with median test accuracy and loss remaining consistent from 95% to 80% PCA compression. The results suggest that the most discriminative signals for loop dependency classification are largely preserved even under significant dimensionality reduction, with only marginal increases in variability observed at the most aggressive compression level.

2.5.4 Statistical Summary of Model Performance

A detailed statistical evaluation assessed the average behavior and consistency of both models under different data compression levels. Tables 3 to 6 present key performance metrics, including mean test accuracy, standard deviation, median, best and worst outcomes, and 95% confidence intervals.

Table 3 – DNN model: mean, standard deviation, and median of test accuracy across 30 runs for different training proportions

Proportion (%)	Mean (%)	Std (%)	Median (%)
95	93.12	3.65	93.91
90	93.62	3.25	94.75
85	94.04	1.66	94.16
80	93.69	3.45	94.41

Source: Prepared by the authors, 2026.

The DNN achieved its highest average accuracy (94.04%) with 85% variance retention, showing the lowest standard deviation (1.66%) and a median of 94.16%, corresponding to a relatively concentrated distribution of results. The close alignment between mean and median values suggests a balanced distribution of results across the 30 executions, as shown in Table 3.

Table 4 – CNN model: mean, standard deviation, and median of test accuracy across 30 runs for different training proportions

Proportion (%)	Mean (%)	Std (%)	Median (%)
95	92.19	2.50	92.50
90	93.09	3.10	94.25
85	92.37	4.69	93.41
80	92.16	3.63	93.08

Source: Prepared by the authors, 2026.

The CNN reached its peak average accuracy (93.09%) with 90% variance retention, with a median of 94.25% but slightly higher variability across runs. Both models maintained competitive performance under moderate dimensionality reduction, showing that classification accuracy remains comparable across different levels of feature compression under the evaluated conditions, as shown in Table 4.

Table 5 – DNN model: best, worst, and 95% confidence interval of test accuracy across 30 runs

Proportion (%)	Best (%)	Worst (%)	95% CI of the Mean
95	95.83	76.17	[91.75, 94.48]
90	96.50	81.33	[92.40, 94.83]
85	96.83	89.00	[93.41, 94.65]
80	96.67	76.33	[92.39, 94.98]

Source: Prepared by the authors, 2026.

The DNN displayed higher consistency at 85% retained variance, with accuracy ranging from 89.00% to 96.83% and a narrow confidence interval [93.41%, 94.65%], as shown in Table 5.

Table 6 – CNN model: best, worst, and 95% confidence interval of test accuracy across 30 runs

Proportion (%)	Best (%)	Worst (%)	95% CI of the Mean
95	96.50	87.33	[91.26, 93.12]
90	96.83	86.33	[91.93, 94.25]
85	97.00	71.00	[90.61, 94.11]
80	96.67	82.17	[90.79, 93.51]

Source: Prepared by the authors, 2026.

The CNN achieved its best individual performance (97.00%) at the same level but exhibited greater dispersion, with the lowest performance dropping to 71.00%, as shown in Table 6.

2.5.5 Best-Case Evaluation of DNN and CNN Models

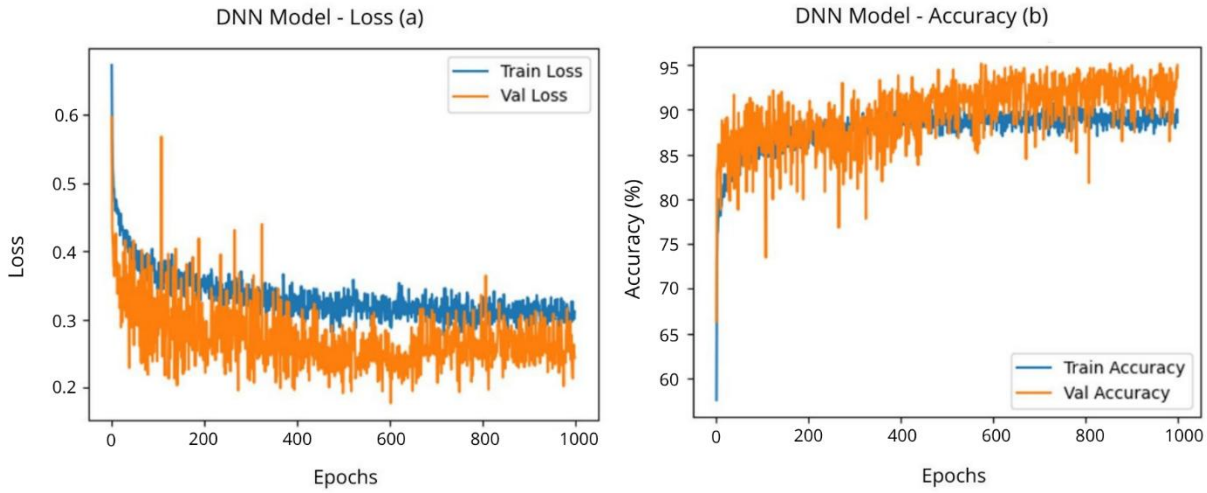
This subsection analyzes the best-performing run from the 30 independent executions for each architecture, including training performance plots, confusion matrices, and classification reports.

The DNN model achieved its best result (96.83% test accuracy) using PCA with 85% variance retention, while the CNN performed best (97.67% test accuracy) without dimensionality reduction, using 100% of features.

2.5.6 DNN Best-Case Performance (PCA 85%)

Figure 9 shows the training behavior of the best DNN model, with training and validation accuracy and loss curves. The model reached final training and validation accuracies of 88.61% and 95.00%, respectively, suggesting a favorable balance between training and validation performance under this configuration.

Figure 9 – Training and validation accuracy and loss of the best DNN model using 85% PCA variance



Source: Prepared by the authors, 2026.

Table 7 presents the confusion matrix of the best-performing DNN configuration, showing a well-balanced classification ability. The model produced only 11 misclassifications for class 0 and 8 for class 1, indicating a low number of misclassifications between the two classes under this experimental setting.

Table 7 – Confusion matrix of the best-performing DNN model

	Predicted 0	**Predicted 1**
Actual 0	312	11
Actual 1	8	269

Source: Prepared by the authors, 2026.

Table 8 complements these findings by providing the detailed classification report. The results confirm consistently high precision, recall, and F1-scores across both classes, with macro and weighted averages of 0.97. These metrics confirm

consistently high precision, recall, and F1-scores across both classes for this execution.

Table 8 – Classification report of the best-performing DNN model

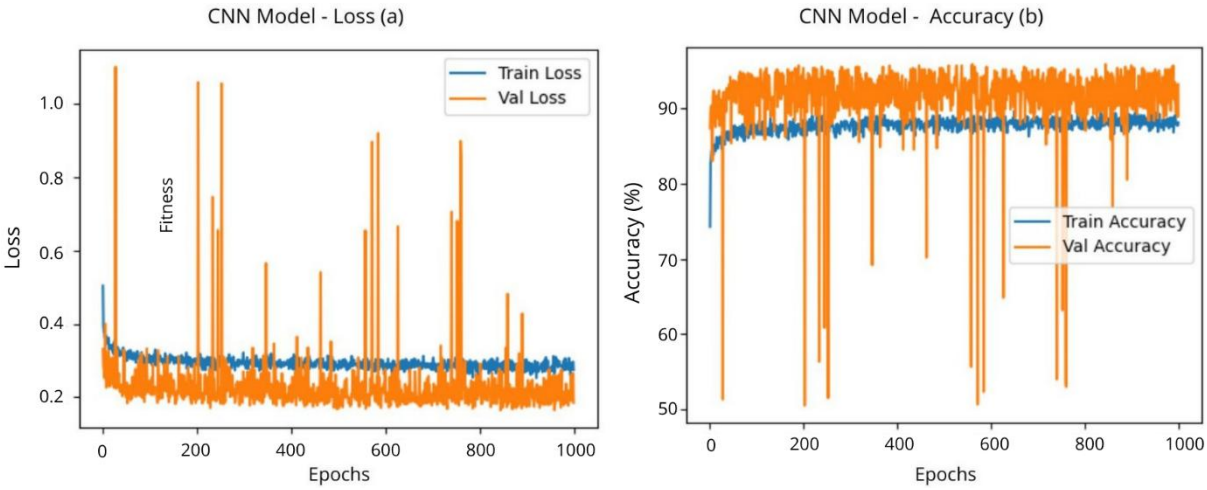
Class	Precision	Recall	F1-score	Support
Dependent Loop (0)	0.97	0.97	0.97	323
Independent Loop (1)	0.96	0.97	0.97	277
Accuracy			0.97	600
Macro avg	0.97	0.97	0.97	600
Weighted avg	0.97	0.97	0.97	600

Source: Prepared by the authors, 2026.

2.5.7 CNN Best-Case Performance (Full Features)

Figure 10 illustrates the learning dynamics of the best-performing CNN model. The network achieved training and validation accuracies of 88.11% and 93.17%, respectively, indicating a consistent improvement across epochs. The close alignment between mean and median values suggests a balanced distribution of results across the 30 executions. Furthermore, the progression of the curves demonstrates that the model continued to refine its feature extraction capability over time, ultimately reaching a steady convergence behavior under the evaluated training configuration.

Figure 10 – Training and validation accuracy and loss of the best CNN model using all features



Source: Prepared by the authors, 2026.

The confusion matrix in Table 9 shows excellent classification performance, with zero false positives for class 0 and only a small number of false negatives in class 1, indicating a low misclassification rate for both classes in this execution.

Table 9 – Confusion matrix of the best-performing CNN model

	Predicted 0	**Predicted 1**
Actual 0	323	0
Actual 1	14	263

Source: Prepared by the authors, 2026.

The classification report for this execution, presented in Table 10, confirms a slightly higher overall accuracy compared to the DNN. Both macro and weighted averages reach 0.98, demonstrating consistent and balanced performance across classes. The high recall for class 0 and strong precision for class 1 emphasize the model's ability to correctly identify each category while minimizing misclassifications. Additionally, the near-equivalence between macro and weighted averages suggests that the CNN handles the class distribution without bias, indicating balanced performance across classes under the evaluated experimental conditions.

Table 10 – Classification report of the best-performing CNN model

Class	**Precision**	**Recall**	**F1-score**	**Support**
Dependent Loop (0)	0.96	1.00	0.98	323
Independent Loop (1)	1.00	0.95	0.97	277
Accuracy			0.98	600
Macro avg	0.98	0.97	0.98	600
Weighted avg	0.98	0.98	0.98	600

Source: Prepared by the authors, 2026.

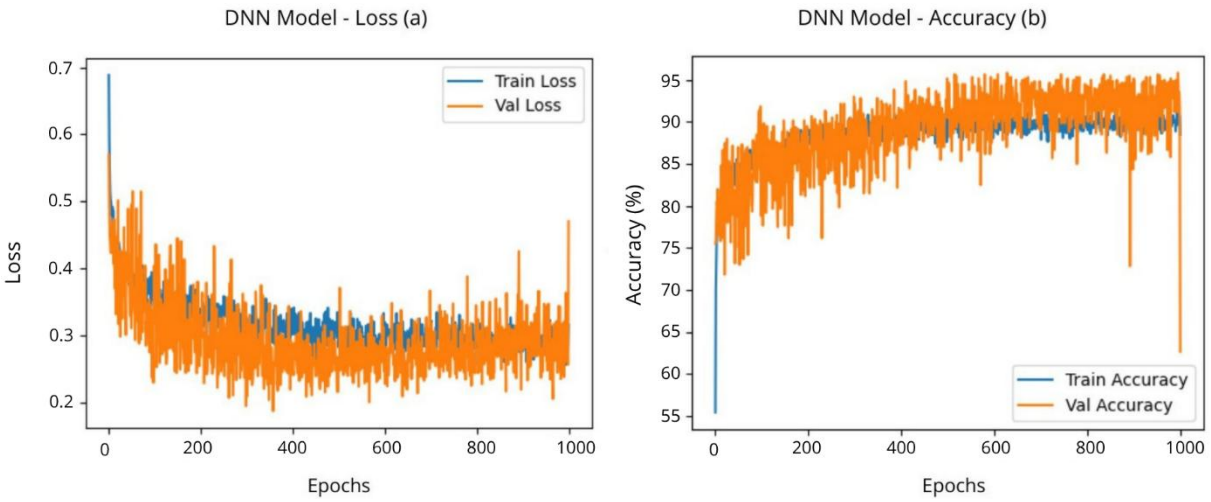
2.5.8 Worst-Case Evaluation of DNN and CNN Models

This section examines the worst-performing runs for both models to understand their limitations and failure scenarios under unfavorable conditions.

2.5.9 DNN Worst-Case Performance (Full Features)

Figure 11 shows the training behavior of the worst-performing DNN model. While it achieved 88.50% training accuracy, validation accuracy dropped to 62.67%, indicating strong overfitting and poor generalization capability. This performance gap highlights the model's sensitivity to particular initializations and its tendency to memorize training patterns rather than learning transferable features.

Figure 11 – Training and validation accuracy and loss of the worst DNN model using all features



Source: Prepared by the authors, 2026.

The confusion matrix in Table 11 reveals severe classification bias, with the model correctly identifying all class 1 instances but misclassifying 75.5% of class 0 samples. This extreme imbalance resulted in the overall 59.33% accuracy.

Table 11 – Confusion matrix of the worst-performing DNN model

	Predicted 0	**Predicted 1**
Actual 0	79	244
Actual 1	0	277

Source: Prepared by the authors, 2026.

This extreme imbalance resulted in the overall 59.33% accuracy reflected in Table 12, where perfect recall for class 1 contrasted sharply with near-failure for class 0. The model's preference for one class severely compromised its discriminative capability, yielding performance metrics that reflect class distribution rather than

meaningful pattern recognition. This behavior reflects a strong bias toward a single class in this execution and indicates sensitivity to initialization and training dynamics.

Table 12 – Classification report of the worst-performing DNN model

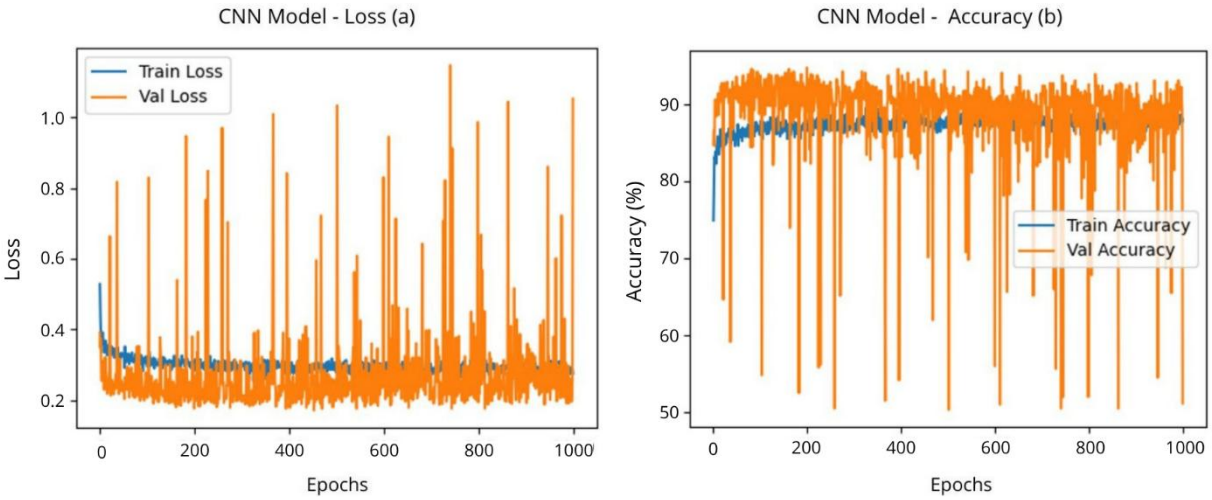
Class	Precision	Recall	F1-score	Support
Dependent Loop (0)	1.00	0.24	0.39	323
Independent Loop (1)	0.53	1.00	0.69	277
Accuracy			0.59	600
Macro avg	0.77	0.62	0.54	600
Weighted avg	0.78	0.59	0.53	600

Source: Prepared by the authors, 2026.

2.5.10 CNN Worst-Case Performance (Full Features)

The worst-performing CNN execution showed a clear generalization gap, with training accuracy at 88.14% and validation accuracy dropping to 51.17% (Figure 12). This 37-point discrepancy indicates severe overfitting, where the model memorized training patterns without learning transferable features. The loss curves reinforce this behavior, displaying a consistent divergence between training and validation as training progresses.

Figure 12 – Training and validation accuracy and loss of the worst CNN model using all features



Source: Prepared by the authors, 2026.

The confusion matrix in Table 13 reveals a clear imbalance in the model's predictions: while it correctly identifies almost all class 0 instances, it fails to generalize to class 1, correctly classifying only 13 examples out of 277.

Table 13 – Confusion matrix of the worst-performing CNN model

	Predicted 0	**Predicted 1**
Actual 0	322	1
Actual 1	264	13

Source: Prepared by the authors, 2026.

The classification report in Table 14 reinforces this imbalance, showing extremely poor performance for class 1, with a recall of only 0.05 and an F1-score of 0.09. The resulting overall accuracy of 55.83% reflects the dominance of class 0 predictions rather than meaningful learning.

Table 14 – Classification report of the worst-performing CNN model

Class	**Precision**	**Recall**	**F1-score**	**Support**
Dependent Loop (0)	0.55	1.00	0.71	323
Independent Loop (1)	0.93	0.05	0.09	277
Accuracy			0.56	600
Macro avg	0.74	0.52	0.40	600
Weighted avg	0.72	0.56	0.42	600

Source: Prepared by the authors, 2026.

These worst-case results highlight the significant variability that can occur across different executions of the same model architecture. While both models achieved excellent performance in their best runs, their worst cases revealed substantial accuracy drops and class imbalance issues, emphasizing the importance of multiple executions for reliable model evaluation.

2.5.11 Qualitative Comparison with Related Work

To contextualize the proposed approach within the broader landscape of learning-based parallelization analysis, Tables 15 and 16 present a qualitative comparison between this work and representative methods discussed in the related literature. Rather than aiming at a direct quantitative benchmark—which would be methodologically inappropriate due to differences in problem formulation, input

representations, and datasets—this comparison highlights key methodological dimensions, including code representation, learning models, evaluation metrics, and computational cost.

Table 15 – Comparison of code representations and learning models used in related work

Work	Representation	Model	Dataset
Shen et al. (2021)	CFG / Graph	GNN	Real-world
Chen et al. (2022)	Multi-view Graph	GNN	Real-world
Mahmud et al. (2025)	GNN + LLM	Transformer	Real-world
Harel et al. (2025)	AST + Structural	Transformer	Real-world
This work	Token sequence	DNN / CNN	Synthetic

Source: Prepared by the authors, 2026.

Table 16 – Comparison of evaluation metrics and computational cost across approaches

Work	Metrics Reported	Variability	Comp. Cost
Shen et al. (2021)	Speedup-based	No	High
Chen et al. (2022)	Task-specific	No	High
Mahmud et al. (2025)	OMPScore	No	Very High
Harel et al. (2025)	Task-specific accuracy	Limited	Very High
This work	Accuracy + CI + KS	Yes	Low

Source: Prepared by the authors, 2026.

As shown in Table 15, most existing approaches rely on rich structural representations such as control-flow graphs, abstract syntax trees, or graph-based intermediate representations, typically combined with graph neural networks or Transformer-based models and evaluated on real-world codebases. In contrast, the proposed method deliberately restricts its input to raw token sequences and employs lightweight DNN and CNN architectures trained on synthetically generated loops.

Table 16 further illustrates that prior work often evaluates performance using task-specific metrics such as speedup or code-quality scores (e.g., OMPScore), frequently without reporting statistical variability across multiple runs. By comparison, this work reports classification accuracy alongside confidence intervals and statistical

significance testing across repeated executions, while maintaining substantially lower computational cost.

This qualitative analysis highlights the distinct positioning of the proposed approach: rather than competing with structure-aware, high-cost models, it explores the feasibility and stability of loop-level parallelizability detection under minimal representational and architectural assumptions. These results complement existing methods by clarifying trade-offs between representation complexity, evaluation rigor, and computational efficiency.

Overall, these results indicate that, within the controlled experimental setting adopted in this study, lightweight neural architectures trained on minimal token-based representations can capture signals associated with loop-carried dependencies. Despite variability across executions, both models achieved high classification accuracy across multiple configurations, in line with the experimental objective of evaluating minimal representational and architectural requirements under simplified conditions.

2.6 DISCUSSION

The experimental results indicate that lightweight neural architectures can effectively distinguish between independent and dependent loops using only token-based representations. Both DNN and CNN models demonstrated consistent performance across 30 executions, with the CNN showing marginally superior mean accuracy while maintaining comparable variability. This suggests that simplified token representations—without complex code structures like ASTs or control-flow graphs—contain sufficient surface-level patterns for loop parallelizability classification within our dataset domain. Rather than competing with compiler-grade dependence analyses or structure-aware learning models, these findings clarify the lower bound of representational complexity required for loop-level parallelizability classification under controlled conditions.

The PCA analysis reveals important insights into the representational requirements of this task. While full-dimensional inputs generally yielded optimal accuracy, models trained on reduced-dimensional variants (preserving 95%–80% variance) maintained competitive performance, demonstrating robustness to input compression. This supports the hypothesis that only a subset of token-space variance

is critical for identifying iteration independence. However, the performance degradation under stronger compression underscores the sensitivity of token-based models to dimensionality reduction that may disrupt sequential patterns. This observation reinforces the central premise of this work, namely that loop parallelizability signals can be captured without relying on full-dimensional or highly structured program representations.

The examination of best- and worst-case executions highlights model stability as a primary concern rather than peak accuracy. The observed variability appears predominantly influenced by random initialization and training stochasticity, consistent with known behavior of shallow neural networks in high-dimensional discrete spaces. Future improvements may require architectural refinements, stronger regularization strategies, or hybrid representations that incorporate limited structural context, particularly to mitigate sensitivity to initialization observed across independent executions.

Collectively, these findings demonstrate that, within a controlled experimental setting, lightweight neural architectures operating on token-based representations can capture meaningful signals associated with loop-carried dependencies. It is important to note that the binary classification adopted in this study abstracts away finer-grained dependency types (e.g., conditional or partial loop-carried dependencies) that commonly arise in real-world programs and remain outside the scope of the present analysis. While this approach does not aim to replace static dependence analysis or structure-aware learning models, it provides empirical evidence that a significant portion of loop parallelizability information is already present at the lexical level, offering insights into the trade-offs between representational simplicity, computational cost, and analytical power.

2.7 CONCLUSION

This study demonstrates that lightweight deep learning models can effectively identify parallelizable loops using only token-based code representations. Through extensive evaluation on synthetic datasets containing independent and dependent loops, both DNN and CNN architectures achieved high accuracy with consistent performance across multiple executions. Statistical analysis revealed comparable effectiveness between models, with the CNN exhibiting marginally superior loss

characteristics but similar accuracy distributions. Rather than proposing a new parallelization algorithm, this work contributes empirical evidence toward understanding the minimum representational and architectural complexity required to distinguish loop-level parallelizability signals under controlled experimental conditions.

Our findings indicate that simple neural architectures can capture meaningful surface-level patterns indicative of loop independence without requiring structural program representations like ASTs or control-flow graphs. The robustness observed under moderate PCA-based dimensionality reduction further suggests that only a subset of token-space variance is essential for this classification task.

Despite these promising results, the approach presents intrinsic limitations stemming from its design choices. The binary classification of loops as independent or dependent simplifies the broader concept of parallelizability and may not fully reflect nuanced real-world scenarios. Additionally, relying on synthetic Python loops constrains the expressiveness of the dataset, which lacks complex dependency interactions, aliasing behaviors, and multi-language characteristics typically found in production environments. Token-based representations further restrict the model's ability to capture structural and semantic aspects of code, limiting the depth of learned patterns. Consequently, the applicability of the proposed method should be interpreted as preliminary, with generalization to real-world code requiring more expressive representations and diverse datasets.

Future research directions include incorporating real-world code benchmarks, exploring transformer-based and graph neural network architectures, and assessing practical deployment in open-source software environments. Additionally, extending the classification framework beyond the binary dependency scheme to handle finer-grained dependency categories, such as conditional dependencies, loop-independent dependencies, and complex data-flow interactions, represents a natural next step toward broader applicability. Such advancements would strengthen the viability of lightweight deep learning approaches for automated parallelization analysis in diverse programming contexts.

2.8 DATA AVAILABILITY

The data and code supporting the findings of this study are publicly available to ensure transparency and reproducibility. All datasets, data generation scripts, preprocessing utilities, and model implementations used in this work are accessible in a public GitHub repository.[3]

The repository provides the complete experimental pipeline required to reproduce the reported results, including scripts for synthetic loop generation using a genetic algorithm, token-based preprocessing and feature mapping, optional dimensionality reduction via PCA, and training and evaluation code for the DNN and CNN models. The project is organized into modular directories corresponding to each stage of the workflow, and a detailed replication guide is provided in the repository README.

All data are synthetically generated and contain no sensitive or proprietary information. There are no access restrictions, and the repository is freely available for research and educational use, fully supporting the reproducibility of the experiments presented in this paper.

[3] https://github.com/IzavanCorreia/DSPPUDNN-Article

3 ARTICLE 2 – AUTOMATIC IDENTIFICATION OF PARALLELIZABLE LOOPS USING TRANSFORMER-BASED SOURCE CODE REPRESENTATIONS

3.1 ABSTRACT

Automatic parallelization remains a challenging problem in software engineering, particularly in identifying code regions where loops can be safely executed in parallel on modern multi-core architectures. Traditional static analysis techniques, including dependence analysis and polyhedral models, often struggle with irregular, non-affine, or dynamically structured code. In this work, we propose a Transformer-based approach for the automatic classification of parallelization potential in source code, focusing on distinguishing independent (parallelizable) loop constructs from undefined ones. We adopt DistilBERT, a distilled and computationally efficient variant of BERT, to directly process source code sequences with subword tokenization, thereby capturing contextual syntactic and semantic patterns without requiring handcrafted features or complex preprocessing. To evaluate the effectiveness of this approach, we constructed a balanced dataset combining synthetically generated loops via evolutionary algorithms with manually annotated real-world code samples, and performed a rigorous 10-fold cross-validation evaluation using multiple performance metrics including accuracy, precision, recall, F1-score, and false positive rate (FPR). The experimental results demonstrate consistently high performance across all folds, with mean test accuracy exceeding 99% and a very low FPR, indicating the model's robustness and reliability for parallelization classification. Comparative analysis shows that the proposed method simplifies the preprocessing pipeline and improves generalization relative to previous token-based models while maintaining computational efficiency. Overall, the findings suggest that lightweight Transformer architectures such as DistilBERT constitute a promising and practical solution for identifying parallelization opportunities in source code within the defined scope of loop-level analysis.

Keywords: Automatic parallelization; Source code analysis; Transformer models; DistilBERT; Loop independence classification.

3.2 INTRODUCTION

Automatic parallelization remains a key challenge in software engineering and compiler optimization, as it directly impacts application performance on modern multi-core and heterogeneous architectures. Traditional static analysis techniques, such as dependence analysis and polyhedral frameworks, rely on precise code representations including abstract syntax trees (ASTs) and control-flow graphs to determine whether loop iterations can be executed in parallel without conflicts (Zinenko et al., 2018; Pouchet et al., 2009). However, these approaches often face limitations when dealing with irregular, non-affine, or highly dynamic code structures, which are common in real-world software systems.

Machine learning and deep learning have emerged as promising alternatives for source code analysis by learning latent representations directly from raw code, reducing the need for handcrafted rules and domain-specific heuristics (Allamanis et al., 2018). Early approaches primarily relied on token-level or sequence-based representations combined with shallow classifiers, while more recent methods leverage deep neural architectures capable of capturing richer syntactic and semantic properties of code. These techniques have shown effectiveness across a wide range of software engineering tasks, including code summarization, defect prediction, clone detection, and program classification (Allamanis et al., 2016; White et al., 2016).

More recently, transformer-based models have significantly advanced representation learning for both natural language and source code through self-attention mechanisms that enable the modeling of long-range dependencies within sequences (Vaswani et al., 2017). Pre-trained transformer models such as CodeBERT and GraphCodeBERT adapt this architecture to programming languages by jointly modeling natural language and code tokens, as well as incorporating data-flow information, achieving state-of-the-art performance in tasks such as code search, clone detection, and program classification (Feng et al., 2020; Guo et al., 2020). Furthermore, transformer architectures have also been explored in contexts directly related to parallelization, demonstrating their potential to classify code regions suitable for OpenMP-based parallel execution (Chen et al., 2024).

Despite these advances, many existing approaches for detecting parallelizable code still rely on structured representations, such as ASTs or graph-based embeddings (Chen et al., 2022; Shen et al., 2021; Guo et al., 2020), which require

complex preprocessing pipelines and substantial domain expertise. Previous work from this research group proposed a lightweight approach based on deep neural networks (DNNs) and convolutional neural networks (CNNs), trained on token sequences generated by a Python preprocessing script, to distinguish independent (parallelizable) loops from dependent ones in synthetic datasets (Correia et al., 2025). While the results demonstrated that token-level representations contain relevant information for parallelism classification, the approach required external preprocessing steps and dimensionality reduction techniques such as Principal Component Analysis (PCA), and it did not exploit contextual representations of code.

In this work, we extend that foundation by leveraging transformer-based contextual representations to improve the automatic classification of parallelization potential in source code. Specifically, we adopt DistilBERT[4], a distilled and computationally efficient variant of BERT that preserves much of the representational power of larger transformer models while reducing model size and inference cost (Sanh et al., 2019). DistilBERT produces contextualized embeddings that capture both syntactic patterns and semantic relationships directly from raw code sequences, enabling effective learning without extensive manual feature engineering and making it suitable for scenarios with constrained computational resources when compared to larger code-specific models.

To assess the effectiveness of the proposed approach, we constructed a balanced dataset combining synthetically generated loops and manually annotated real-world code samples, and evaluated the model using rigorous 10-fold cross-validation. In addition to overall accuracy, we emphasize metrics such as the false positive rate (FPR), which is particularly relevant in parallelization scenarios where incorrectly classifying dependent code as parallelizable may lead to unsafe or inefficient execution. The experimental results indicate that the transformer-based model achieves stable and competitive performance across multiple evaluation metrics, while simplifying the preprocessing pipeline and improving generalization when compared to the previous token-based approach.

The remainder of this paper is organized as follows: Section 3.3 reviews related work on parallelization and deep learning for code analysis, Section 3.4 describes the proposed method, Section 3.5 presents the experimental evaluation, Section 3.6

[4] DistilBERT Documentation: https://huggingface.co/docs/transformers/model_doc/distilbert

discusses the findings, and Section 3.7 concludes the paper and outlines directions for future work.

3.3 RELATED WORKS

Research on automatic parallelization has traditionally been dominated by compiler-based static analysis techniques. Classical approaches rely on dependence analysis, control-flow analysis, and polyhedral models to determine whether loop iterations can be safely executed in parallel (Pouchet et al., 2009; Zinenko et al., 2018). While these methods provide strong theoretical guarantees, they require affine loop bounds and regular memory access patterns, which significantly limits their applicability to real-world programs that exhibit dynamic control flow, pointer aliasing, and irregular data access patterns.

To overcome these limitations, machine learning techniques have been increasingly explored for source code analysis. Early work focused on learning from token-level representations or handcrafted features extracted from code, demonstrating that statistical models could capture useful properties related to program behavior (Allamanis et al., 2016). More comprehensive surveys highlight the growing adoption of machine learning and deep learning methods across software engineering tasks, including bug detection, clone detection, and program classification, emphasizing their ability to generalize beyond rigid syntactic rules (Allamanis et al., 2018; White et al., 2016).

The introduction of transformer architectures marked a significant advancement in code representation learning. By leveraging self-attention mechanisms, transformers enable the modeling of long-range dependencies within sequences, which is particularly relevant for source code understanding (Vaswani et al., 2017). Building on this paradigm, models such as CodeBERT and GraphCodeBERT incorporate pre-training on large-scale code corpora and, in some cases, explicit data-flow information, achieving state-of-the-art performance on several downstream tasks, including code search, clone detection, and program classification (Feng et al., 2020; Guo et al., 2020). Despite their effectiveness, these models are typically large and computationally demanding, which may limit their applicability in scenarios with constrained resources.

More closely related to this work are studies that apply deep learning techniques to identify parallelizable regions in code. Recent transformer-based approaches, such as PragFormer, have demonstrated the feasibility of using attention-based models to classify code segments according to their suitability for OpenMP parallelization (Chen et al., 2024). These methods highlight the potential of learned representations to support parallelization decisions without relying solely on traditional compiler analyses.

Prior work from this research group proposed a lightweight deep learning approach based on deep neural networks and convolutional neural networks trained on token sequences generated via Python-based preprocessing scripts (Correia et al., 2025). That study demonstrated that even shallow token representations contain sufficient information to distinguish independent from dependent loops. However, the approach required external preprocessing steps and dimensionality reduction techniques, such as principal component analysis, and did not exploit contextual representations capable of modeling long-range dependencies within code.

In contrast to existing methods, the present work leverages a distilled transformer model to balance representational power and computational efficiency. By adopting DistilBERT, the proposed approach captures contextual syntactic and semantic information directly from source code while maintaining a lightweight architecture suitable for practical deployment. This positions the method as a complementary alternative to both traditional static analysis techniques and larger transformer-based models, addressing the need for efficient and accurate identification of parallelization opportunities in modern software systems.

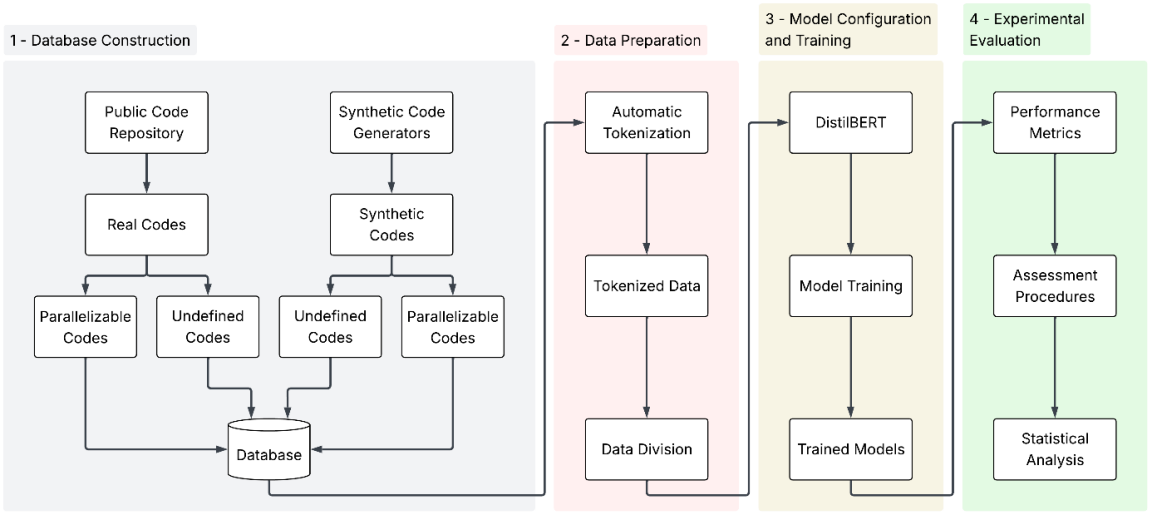
3.4 METHODOLOGY

This section describes the methodology for developing and evaluating the automatic classification system for parallelization potential in programming codes. The process involves five main stages: (1) database construction, (2) data preparation, (3) model setup and training, (4) experimental evaluation, and (5) computational environment. Figure 13 shows the complete pipeline and the flow of each stage.

The process starts with the construction of the dataset using real and synthetic code samples. Next, the data are automatically tokenized and split into training, validation, and test sets. The DistilBERT model is then configured and fine-tuned using

the prepared data. Finally, the trained models are evaluated using performance metrics and statistical analyses.

Figure 13 – Workflow of the proposed methodology



Source: Prepared by the authors, 2026.

3.4.1 DATABASE CONSTRUCTION

The database was carefully built from two complementary sources: automatically generated synthetic codes and real-world codes collected from public repositories.

3.4.2 SYNTHETIC CODE GENERATION

To ensure diversity and control over the classes of interest, we used two generators based on genetic algorithms, following the methodology established in our previous work (Correia et al., 2025). The generators were implemented in Python using the DEAP (Distributed Evolutionary Algorithms in Python) library[5] (DEAP, 2012) and produced two distinct types of codes:

- Class 1 – Parallelizable: Independent loops where each iteration has no data dependencies with other iterations, allowing parallel execution without conflicts.
- Class 0 – Undefined: Loops with non-trivial dependencies or whose parallelization analysis requires complex considerations, including cases where parallelization is not possible or cannot be straightforwardly determined.

[5]DEAP documentation: https://deap.readthedocs.io/en/master/

The fitness function adopted in the evolutionary generator follows the formulation introduced in Correia et al. (2025). For completeness, the equation is reproduced below, while the detailed derivation and motivation are presented in the previous study.

$$f(p) = \sum_{i=1}^{7} s_i \; + \; s_{comp} \quad (1)$$

The score is computed as the sum of rewards and penalties linked to structural characteristics of the generated program. Each component promotes realistic software elements, including imports, functions, conditionals, variable usage, and code length, while stricter penalties target the number of loops and compilation success. This additive formulation drives the search toward syntactically valid and sufficiently complex programs while preserving variability across samples.

The evolutionary process used a population of 10,000 individuals across 50 generations, with a crossover rate of 0.9 and a mutation rate of 0.1. At the end of the procedure, 8,000 synthetic samples were produced, and the distribution between independent and dependent loops emerged from the evolutionary dynamics.

3.4.3 REAL CODE COLLECTION AND ANNOTATION

To complement the synthetic data and enhance the database's representativeness, we collected 340 real code samples from public GitHub repositories and examples available on the internet. Each real code sample underwent a thorough manual analysis conducted by the authors and was classified using the same criteria defined for synthetic codes:

- Class 1 – Parallelizable;
- Class 0 – Undefined.

This manual analysis ensured the quality of the annotations and the relevance of the examples for the classification task. The relatively limited volume of real samples reflects the inherent cost of this annotation process: unlike synthetic generation, which can be scaled automatically, manual classification of real code requires careful individual inspection of each loop instance to determine its dependency profile, making large-scale collection infeasible within the scope of this work. Consequently, the dataset was not naturally balanced between synthetic and real samples, as the

proportion of real code was constrained by the practical limits of manual annotation effort.

3.4.4 FINAL DATABASE COMPOSITION

The final database totaled 8,340 examples, consisting of:

- 4,140 parallelizable codes: comprising 4,000 synthetic samples and 170 real samples;
- 4,140 undefined codes: comprising 4,000 synthetic samples and 170 real samples.

After balancing the dataset, both classes (Parallelizable and Undefined) are now equally represented, ensuring a more uniform learning process and reducing potential bias during training.

No resampling techniques were required in the final configuration, as the balanced distribution itself provided a robust foundation for model training. With this updated dataset, the models achieved improved and more stable performance when compared to the configuration without resampling, indicating that class balance contributed positively to predictive reliability.

This research assumes that the synthetically generated loops cover representative patterns of the repetition structures observed in real codes. More complex loop transformations—such as fusion, unrolling, or tiling—were not included, as these variations fall outside the scope of this work.

3.4.5 DATA PREPARATION

Unlike the previous work (Correia et al., 2025), which relied on an external Python script for automated tokenization and required dimensionality reduction via Principal Component Analysis (PCA) (Pearson, 1901; Gray, 2017), the selected Transformer model processes source code directly as text, significantly simplifying the preparation pipeline.

3.4.6 AUTOMATIC TOKENIZATION

We employed the tokenizer from the DistilBERT model (Sanh et al., 2019) to convert source codes into sequences of numerical tokens. The tokenization parameters were configured as follows:

- Maximum length: 512 tokens;
- Truncation: enabled for exceeding sequences;
- Padding: enabled to uniformize sequence length;
- Attention masks: automatically generated to distinguish real tokens from padding.

This integrated tokenization approach leverages the model's pre-trained subword vocabulary, which provides a more natural representation of programming language constructs compared to the traditional deep learning models like DNNs and CNNs in our previous work (Correia et al., 2025). The subword tokenization effectively captures common programming patterns while maintaining the ability to process unseen tokens through their constituent parts.

3.4.7 DATA SPLITTING

To ensure robust evaluation and prevent overfitting, we employed 10-fold cross-validation using the Scikit-learn implementation (Scikit-learn). For each fold, the data was subdivided as follows:

- Training set: 64% of the data;
- Validation set: 16% of the data;
- Test set: 20% of the data.

This partitioning was achieved through a two-step process within each fold: first separating the test set (20%), then dividing the remaining 80% into training (64%) and validation (16%) subsets. This approach maintained consistent class distributions across all splits while preserving the independence of the test set throughout the cross-validation process.

3.4.8 MODEL SETUP AND TRAINING

3.4.8.1 MODEL SELECTION

We selected the DistilBERT model, a lighter and more efficient Transformer architecture that maintains advanced contextual understanding capabilities. This choice was based on the following criteria:

- Computational efficiency: distilled version of BERT, suitable for iterative experiments;

- Context capture: ability to understand complex relationships in structured language, such as source code;
- Direct processing: eliminates the need for complex preprocessing and manual feature engineering;
- Pre-training: leverages linguistic knowledge acquired during pre-training on large corpora.

The model was loaded from the Hugging Face Transformers library[6] (Wolf et al., 2020) using the distilbert-base-uncased version.

3.4.8.2 ARCHITECTURE AND HYPERPARAMETERS

We configured the model for binary classification with the following specifications:

- Number of labels: 2 (Parallelizable, Undefined);
- Number of epochs: 50;
- Batch size: 16;
- Optimizer: AdamW (Loshchilov & Hutter, 2017);
- Learning rate: 2e-5;
- Learning rate scheduler: linear with warmup;
- Loss function: Binary Cross-Entropy (Goodfellow et al., 2016);
- Early stopping: patience of 50 epochs.

These hyperparameters were defined after a series of preliminary tests with different configurations, where we explored various parameter combinations. The selected configuration proved to be the most suitable for the task, demonstrating stability during training and good generalization performance, with no evidence of overfitting or convergence issues.

3.4.8.3 TRAINING PROCESS

The training was conducted using the Trainer class from the Transformers library, with the following configurations:

- Evaluation strategy: every epoch;
- Saving strategy: every epoch;
- Metric for best model: lowest validation loss;
- Loading best model at end: enabled;

[6] Documentation Transformers: https://huggingface.co/docs/transformers/index

- Save limit: 1 checkpoint;
- Early stopping callback: EarlyStoppingCallback with patience of 50 epochs.

For each of the 10 folds, the model was reinitialized and trained from scratch, ensuring independence between executions. The early stopping callback was configured with a patience of 50 epochs, serving as a contingency mechanism while allowing complete training cycles. The best model for each fold was automatically selected based on validation loss and loaded at the end of training.

3.4.9 EXPERIMENTAL EVALUATION

3.4.9.1 PERFORMANCE METRICS

We evaluated the model using multiple metrics to obtain a comprehensive performance overview:

- Accuracy: proportion of correct classifications;
- Precision: ability to avoid classifying non-parallelizable loops as parallelizable;
- Recall: ability to identify all parallelizable loops;
- F1-score: harmonic mean between precision and recall;
- Confusion matrix: detailed analysis of classification errors.

All metrics were calculated using Scikit-learn functions, with binary averaging for precision, recall, and F1-score.

3.4.9.2 EVALUATION PROCEDURES

For each of the 10 folds, we performed the following analyses:

- Learning curves: plots of loss and accuracy during training;
- Per-class metrics: precision, recall, and F1-score for each class;
- Confusion matrices: analysis of error patterns;
- Cross-fold comparison: stability and consistency of performance across folds.

The evaluation included comprehensive analysis of all 10 trained models, with detailed comparison of training, validation, and test performance for each fold.

3.4.9.3 STATISTICAL ANALYSIS

Data analysis was performed through a dedicated Python workflow that integrated specialized libraries for the calculation of descriptive statistics and confidence intervals. For each metric—training accuracy, validation accuracy, and test accuracy—we computed the mean, sample standard deviation, median, and the 95%

confidence interval using the Student's *t*-distribution over the sample means (Student, 1908). These metrics were obtained from the ten executions corresponding to the ten folds used in the experimental procedure, ensuring consistency with the overall evaluation protocol.

To visually characterize the variability of the results, boxplots were generated for all accuracy metrics, including the explicit representation of potential outliers when present. Complementarily, histograms were plotted to examine the distribution shape of the accuracy values across the folds.

Beyond accuracy, the analysis also included the mean, standard deviation, median, and 95% confidence intervals for the F1-score, precision, and recall, computed over the ten folds using the same Student's *t*-distribution approach (Student, 1908).

A detailed analysis of the validation error was also performed across all folds. For each fold, we computed the mean, standard deviation, median, and the 95% confidence interval of the validation loss, measured using the binary cross-entropy function. This evaluation was conducted because the validation error constitutes one of the key criteria used to identify the best and worst-performing models—specifically, the model exhibiting the *lowest* validation error.

In an analogous manner, a comprehensive statistical analysis of the false positive rate (FPR) was conducted across all folds. For this metric, we computed the mean, standard deviation, median, and the 95% confidence interval using the same Student's *t*-distribution procedure applied to the validation error. This analysis enables a robust assessment of the model's tendency to incorrectly classify instances from the Undefined class (negative) as Parallelizable (positive), which represents a critical error in the context of automatic parallelization.

Following this analysis, we proceed to the identification of the best and worst cases according to the two selection criteria adopted in this study. The first criterion selects the model with the lowest validation error (best case) and the model with the highest validation error (worst case). The second criterion focuses on the model's behavior regarding the critical error for our application: the false positive rate (FPR), defined as the proportion of instances from the Undefined class (negative) that are incorrectly predicted as Parallelizable (positive). Under this criterion, the best model is

the one that presents a zero or minimal FPR, while the worst model is the one exhibiting the highest FPR.

For each selected model—best and worst under each criterion—we present the results in the following order:

1. Training and validation curves;
2. Confusion matrix;
3. Classification report.

This structured evaluation enables a thorough inspection of model behavior under both performance-oriented and application-critical perspectives.

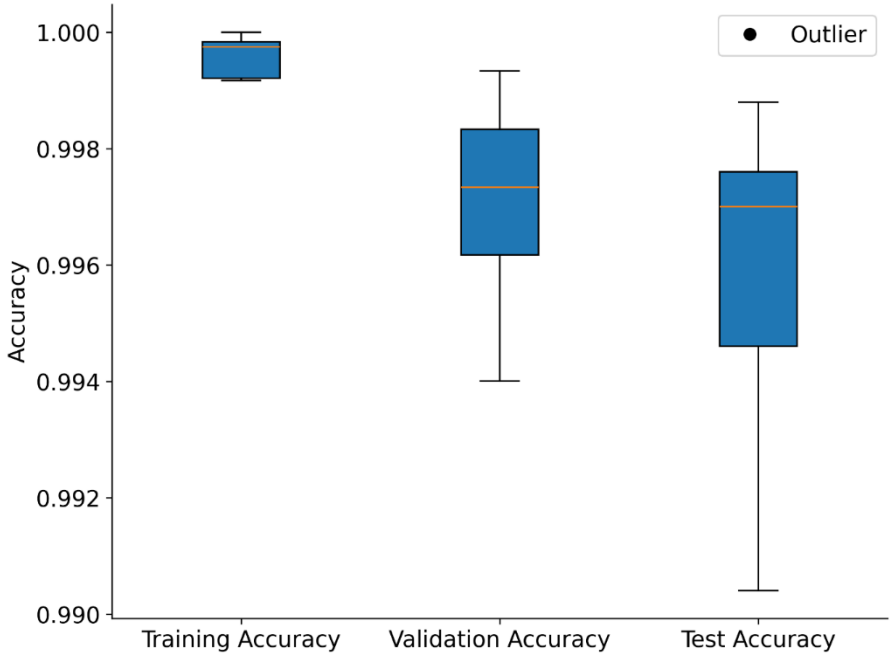
3.5 RESULTS

This section provides a comprehensive analysis of the performance of the DistilBERT model in classifying parallelizable and undefined loops. The results are organized progressively, beginning with an overview of the model's average behavior across all folds, followed by a detailed analysis of accuracy. Subsequent subsections present the remaining performance metrics—precision, recall, and F1-score—in a dedicated evaluation. The section also incorporates two independent criteria for selecting representative models: the validation error and the number of false positives. Because these criteria assess different and equally relevant aspects of performance, four models are examined in the best- and worst-case analyses: the model with the lowest validation error and the one with the lowest number of false positives (best cases), as well as the model with the highest validation error and the one with the highest number of false positives (worst cases). The section concludes with a discussion of the results and the model's robustness. All evaluations were conducted using 10-fold cross-validation, ensuring statistically robust and reliable analysis.

3.5.1 AVERAGE MODEL BEHAVIOR

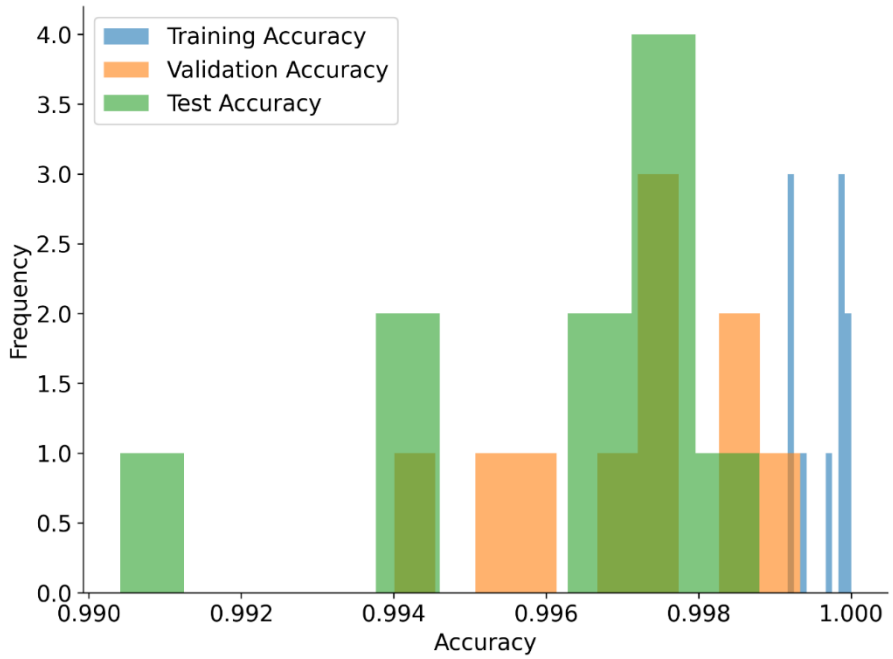
To assess the average behavior of the DistilBERT model across all cross-validation folds, we analyzed the accuracy distributions for the training, validation, and test sets. Figure 14 presents a boxplot that statistically summarizes these distributions, while Figure 15 shows the corresponding histograms, providing a more detailed visualization of accuracy frequency.

Figure 14 – Boxplot of training, validation, and test accuracies for the 10 cross-validation folds



Source: Prepared by the authors, 2026.

Figure 15 – Histograms of training, validation, and test accuracies for the 10 folds



Source: Prepared by the authors, 2026.

The results show that the model achieved high and highly consistent performance across all evaluated sets. The mean training accuracy was approximately 0.99965, indicating excellent ability to fit the data. The mean validation accuracy, around 0.99753, demonstrates that the model maintains very high performance even on unseen data, with low variation across folds. The mean test accuracy, equal to 0.99600, reinforces the model's robustness and the absence of overfitting, as validation and test performances remain close and at high levels. The small dispersion observed in both the boxplot and histograms highlights the stability of DistilBERT for the proposed task.

3.5.2 OVERALL ACCURACY PERFORMANCE

For a more detailed quantitative analysis of the model's performance, Table 17 presents the main consolidated statistical metrics of accuracy obtained in the training, validation, and test sets across the 10 cross-validation folds.

Table 17 – Statistical metrics for training, validation, and test accuracies across the 10 cross-validation folds

Set	Mean	Std. Deviation	Median	95% CI of the Mean
Training	0.99965	0.00030	0.99983	0.99944–0.99986
Validation	0.99753	0.00109	0.99784	0.99675–0.99831
Test	0.99600	0.00261	0.99650	0.99413–0.99787

Source: Prepared by the authors, 2026.

The results demonstrate exceptionally high and consistent performance across all sets. The mean training accuracy of 99.96% confirms the model's ability to effectively learn the patterns present in the data. The mean validation (99.75%) and test (99.60%) accuracies show excellent generalization capability, maintaining very close and high-level results.

The analysis of the standard deviation reveals low variability across folds, reinforcing the model's stability. The training set exhibited the lowest variability (SD = 0.00030), followed by validation (SD = 0.00109) and test (SD = 0.00261). The proximity between means and medians indicates balanced distributions with no relevant skewness.

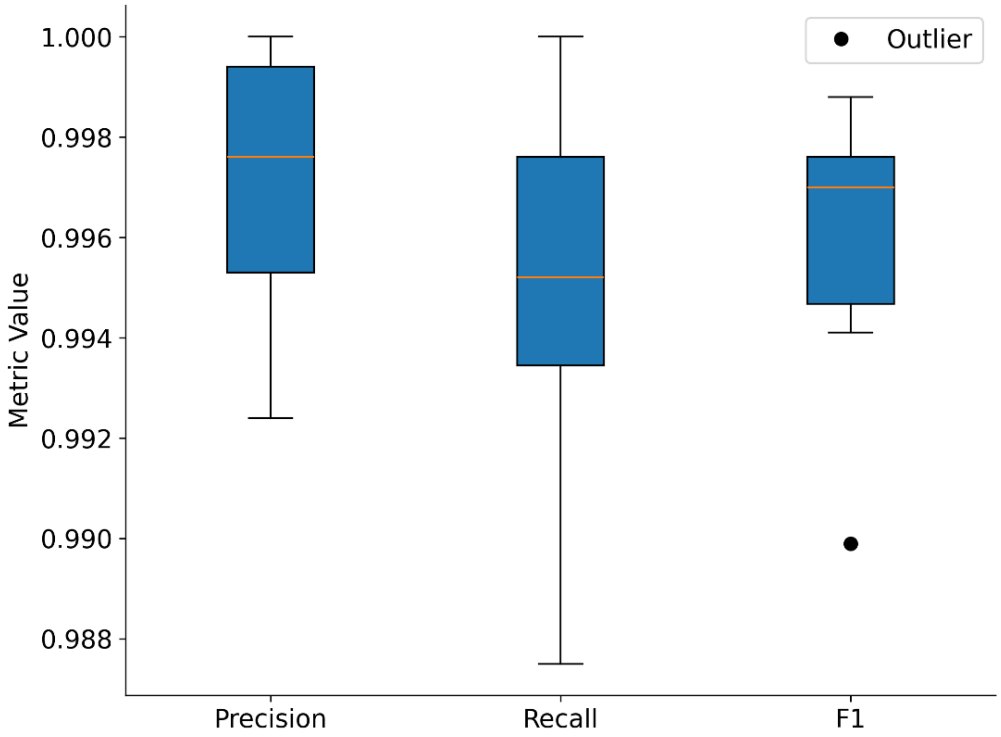
The 95% confidence intervals, calculated using Student's t-test, provide a robust estimate of mean precision. For the test set, for example, we can state with 95% confidence that the mean accuracy lies between 99.41% and 99.78%. For validation, the interval ranges between 99.67% and 99.83%. These narrow intervals further reinforce the strong statistical reliability of the results.

The combination of high mean accuracies, low variability, and narrow confidence intervals confirms the robustness of DistilBERT for the classification task, showing that the model consistently maintains its performance when exposed to new data.

3.5.3 PERFORMANCE ON PRECISION, RECALL, AND F1-SCORE

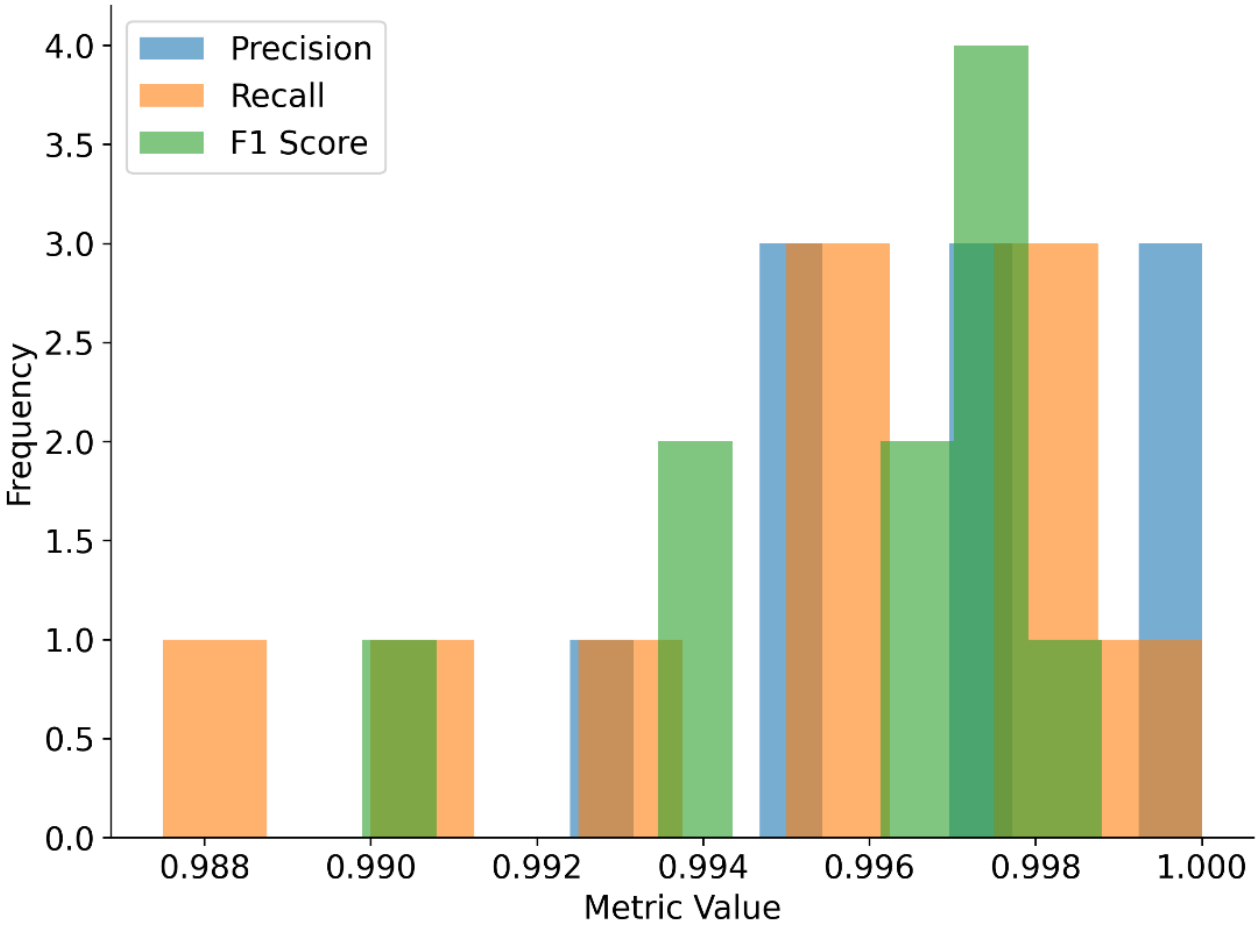
To complement the analysis of the model's performance, we evaluated the precision, recall, and F1-score metrics on the test set. Figure 16 presents the distribution of these metrics using boxplots, while Figure 17 shows the corresponding histograms, allowing for a detailed visual analysis of the behavior of the three metrics across the 10 cross-validation folds.

Figure 16 – Boxplots of precision, recall, and F1-score metrics across the 10 cross-validation folds



Source: Prepared by the authors, 2026.

Figure 17 – Histograms of precision, recall, and F1-score metrics across the 10 cross-validation folds.



Source: Prepared by the authors, 2026.

The visualizations show that all metrics maintain consistently high values, with most observations concentrated above 0.990. Recall exhibits a slightly more compact distribution, indicating stable performance in correctly identifying positive examples. Precision, although exhibiting slightly greater variation across folds, remains consistently high in all cases. The F1-score, as a balanced metric between precision and recall, reflects an intermediate behavior while maintaining high stability across the folds.

For a more detailed quantitative analysis, Table 18 presents the consolidated statistics for the three metrics.

Table 18 – Statistical metrics for Precision, Recall, and F1-Score on the test set

Metric	Mean	Std. Deviation	Median	95% CI of the Mean
Precision	0.9971	0.0026	0.9976	0.9953–0.9989
Recall	0.9949	0.0038	0.9952	0.9922–0.9976
F1-Score	0.9960	0.0026	0.9970	0.9941–0.9979

Source: Prepared by the authors, 2026.

The numerical results confirm the visual observations from the figures. Precision exhibits the highest mean among the metrics (99.71%) with low variability (SD = 0.0026), indicating stability in the model's ability to avoid false positives. Recall, with a mean of 99.49%, also demonstrates strong performance in identifying true positives, although with slightly higher variability (SD = 0.0038). The F1-score, with a mean of 99.60%, reflects an appropriate balance between the two previous metrics, capturing both the model’s ability to correctly identify positive examples and its ability to minimize classification errors.

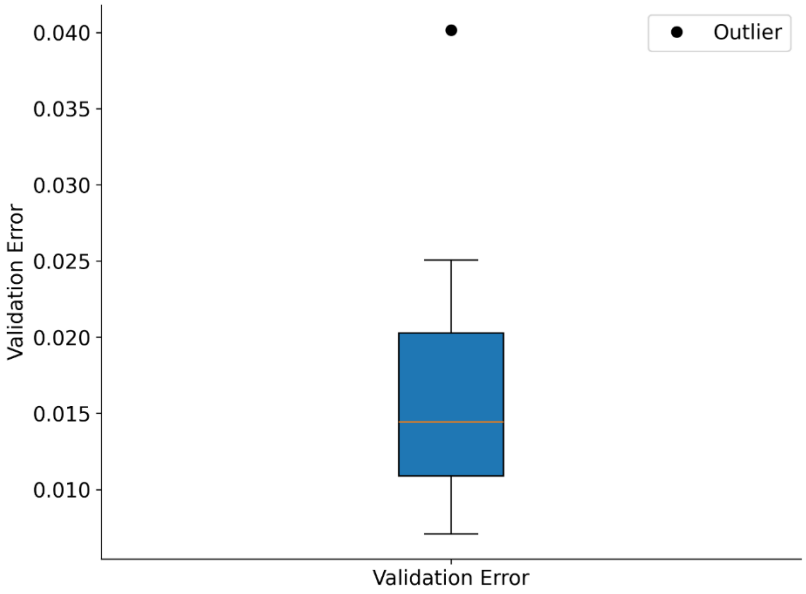
The 95% confidence intervals, computed using Student’s t-test, reinforce the reliability of the results. For the F1-score, the population mean lies between 99.41% and 99.79%; for recall, between 99.22% and 99.76%; and for precision, between 99.53% and 99.89%. The proximity between means and medians across all metrics indicates symmetrical and consistent distributions across the folds, corroborating the model’s stability.

The high and consistent performance observed across all evaluated metrics—precision, recall, and F1-score—confirms the effectiveness of the DistilBERT model for the loop classification task. The results demonstrate not only high overall accuracy but also a proper balance between the ability to correctly identify positive examples and avoid false positives, reinforcing the robustness of the model.

3.5.4 Analysis of the Validation Error as a Selection Criterion

The criterion used to select the best model in each fold was based on the lowest validation error, employing the *Binary Cross-Entropy* function as the evaluation metric during training. This approach makes it possible to identify the points of best model generalization, avoiding overfitting and ensuring that performance is preserved on unseen data.

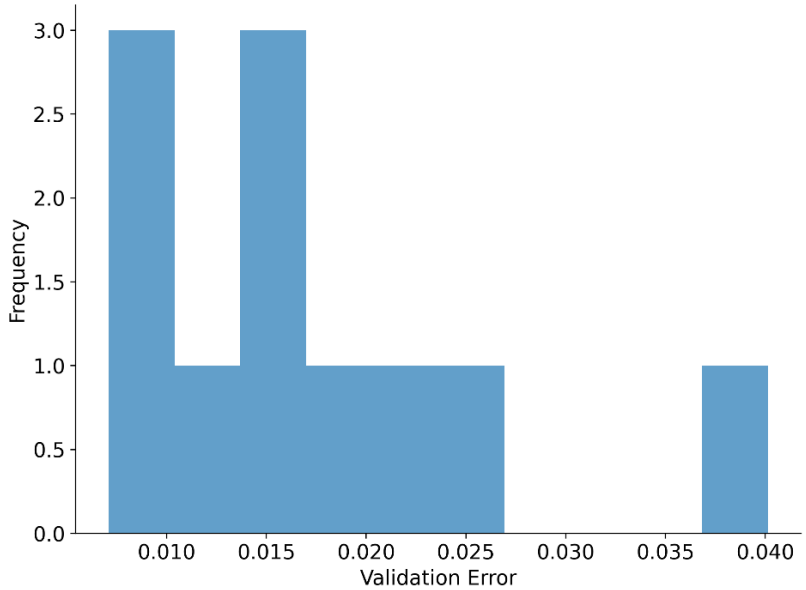
Figure 18 – Boxplot of the validation error (Binary Cross-Entropy) for the 10 folds



Source: Prepared by the authors, 2026.

Figure 18 presents the distribution of the validation error through a boxplot, while Figure 19 shows the corresponding histogram, allowing a detailed visual inspection of the error behavior across the 10 folds.

Figure 19 – Histogram of the validation error (Binary Cross-Entropy) for the 10 folds



Source: Prepared by the authors, 2026.

The visualizations show that the validation error exhibits consistently low values, ranging approximately from 0.007 to 0.040. The boxplot reveals a compact distribution, with most values concentrated in the lower range, indicating good stability across the folds. The histogram reveals an asymmetric distribution with higher density in the lowest error values, which is desirable for an error metric.

For a more detailed quantitative analysis, Table 19 presents the consolidated statistics of the validation error.

Table 19 – Statistical metrics of the validation error (Binary Cross-Entropy)

Metric	Mean	Std. Deviation	Median	95% CI of the Mean
Validation Error	0.01772	0.00965	0.01512	0.01081–0.02462

Source: Prepared by the authors, 2026.

The numerical results confirm the visual observations. The mean validation error of 0.01772, combined with the median of 0.01512, indicates that most folds exhibit very low error values, with a few slightly higher values pulling the mean upward — as reflected by the maximum of approximately 0.040. The standard deviation of 0.00965 reveals moderate variability, also reflected in the extent of the whiskers in the boxplot.

The 95% confidence interval, estimated between 0.01081 and 0.02462, shows that the population mean validation error is concentrated within a narrow range, reinforcing the stability of the validation process.

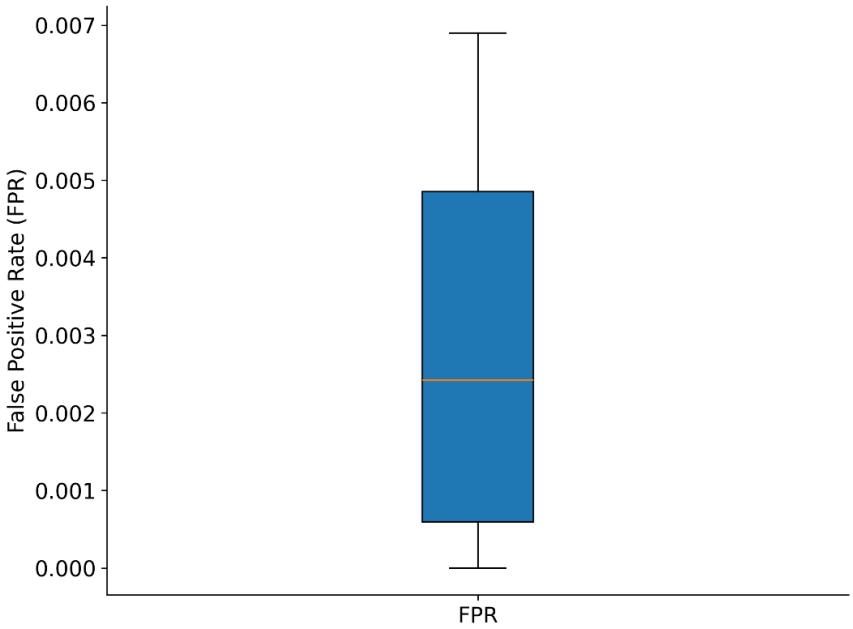
The combination of low absolute error values and the consistency observed across the folds validates the effectiveness of the selection criterion based on the lowest validation error, demonstrating that this approach is suitable for identifying models with strong generalization capability in the proposed classification task.

3.5.5 FALSE POSITIVE RATE (FPR) ANALYSIS

In addition to using the validation error as the selection criterion for the best-performing model, we also conducted a detailed analysis of the False Positive Rate (FPR) across all folds of the cross-validation. The FPR represents the proportion of negative instances incorrectly classified as positive, which is particularly relevant in software defect prediction tasks. A high FPR would lead to false alarms, unnecessarily increasing the inspection effort and causing inefficient allocation of maintenance

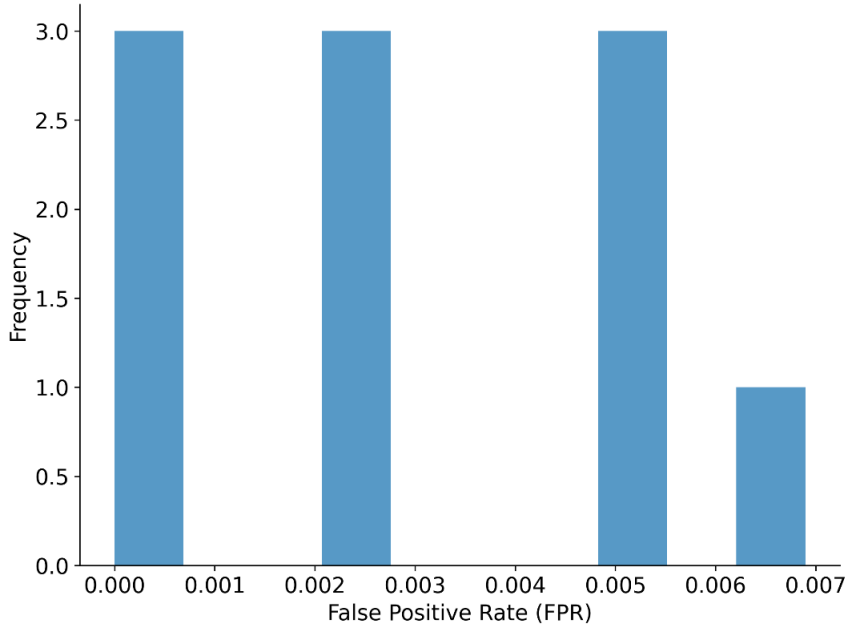
resources. Therefore, examining the statistical behavior of this metric provides further insight into the reliability and robustness of the model.

Figure 20 – Boxplot of the False Positive Rate (FPR) across the 10-fold cross-validation



Source: Prepared by the authors, 2026.

Figure 21 – Histogram of the False Positive Rate (FPR) across the 10-fold cross-validation



Source: Prepared by the authors, 2026.

Figure 20 presents the boxplot of the FPR values. The distribution is highly concentrated near zero, reflecting that the model rarely produces false alarms. Only a few folds exhibit small non-zero values, and even in these cases, the rate remains extremely low. Figure 21 reinforces this observation, showing that most folds lie within the lowest bins of the distribution.

Table 20 – Statistical metrics for the False Positive Rate (FPR) across the 10-fold cross-validation

Metric	Mean	Std. Deviation	Median	95% CI of the Mean
FPR	0.002872	0.002439	0.002418	0.001197–0.004546

Source: Prepared by the authors, 2026.

The results show a mean FPR of 0.002872, with a median of similar magnitude, indicating consistent behavior among the folds. The low standard deviation and the narrow confidence interval further confirm the stability of the model with respect to false positive generation.

3.5.6 ANALYSIS OF THE BEST-PERFORMING MODELS

Based on the two selection criteria previously discussed — the lowest validation error and the lowest critical error (False Positive Rate) — this section presents a detailed analysis of the best-performing models identified during the 10-fold cross-validation process. For each criterion, the selected model is analyzed through its training dynamics, confusion matrix, and classification metrics.

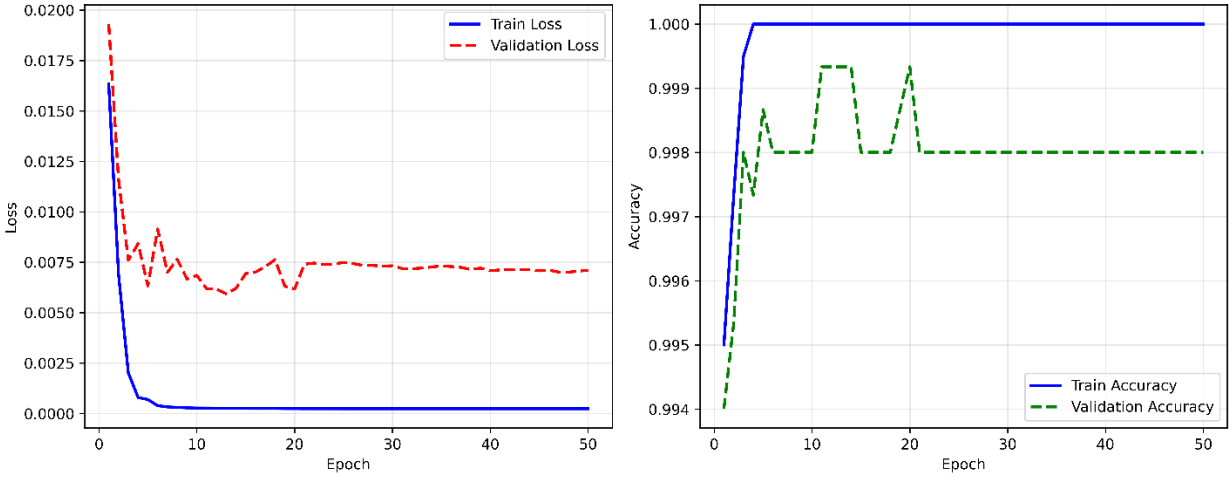
3.5.6.1 BEST MODEL ACCORDING TO THE LOWEST VALIDATION ERROR

Among the ten folds evaluated, **Fold 7** achieved the lowest validation error, making it the best-performing model according to the validation loss criterion. This result indicates superior generalization capability, as the model minimizes the Binary Cross-Entropy loss on unseen validation data.

Figure 22 presents the training and validation loss curves, along with the corresponding accuracy curves. The training loss rapidly converges toward zero, while the validation loss stabilizes at a low value, showing no signs of divergence or instability. Additionally, the training accuracy reaches 100%, and the validation

accuracy remains consistently high throughout the training process, indicating a well-fitted model without evidence of overfitting.

Figure 22 – Training and validation loss and accuracy curves for Fold 7



Source: Prepared by the authors, 2026.

The classification performance of Fold 7 is summarized by the confusion matrix presented in table 21. The results show that the model correctly classified almost all instances, with only a minimal number of misclassifications.

Table 21 – Confusion matrix for Fold 7

Actual / Predicted	Undefined	Independent
Undefined	414	1
Independent	2	417

Source: Prepared by the authors, 2026.

Quantitatively, Fold 7 achieved 417 true positives (TP), 414 true negatives (TN), only 2 false negatives (FN), and 1 false positive (FP). Table 22 presents the corresponding classification metrics. The model achieved a recall of 0.9952, precision of 0.9976, and an F1-score of 0.9964, demonstrating an excellent balance between sensitivity and precision.

Table 22 – Classification report for Fold 7

Metric	Precision	Recall	F1-score
Independent Class	0.9976	0.9952	0.9964

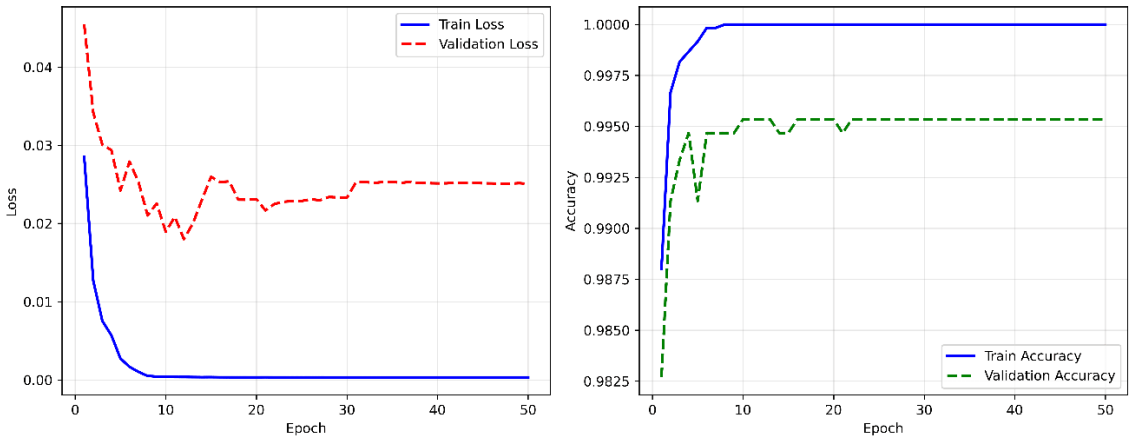
Source: Prepared by the authors, 2026.

3.5.6.2 BEST MODEL ACCORDING TO THE LOWEST CRITICAL ERROR

Considering the False Positive Rate (FPR) as the critical error criterion, **Fold 8** emerged as the best-performing model. This fold achieved the lowest FPR among all evaluated models, which is particularly relevant for the proposed task, as false positives may lead to unnecessary inspections and inefficient allocation of maintenance resources.

Figure 23 presents the training and validation curves for Fold 8. The training process shows fast convergence, with training loss approaching zero and validation loss stabilizing at a low level. The training accuracy reaches 100%, while the validation accuracy remains consistently high, indicating stable learning behavior and strong generalization.

Figure 23 – Training and validation loss and accuracy curves for Fold 8



Source: Prepared by the authors, 2026.

The confusion matrix for Fold 8 is presented in table 23. Notably, this model produced no false positives, which directly explains its selection under the critical error criterion.

Table 23 – Confusion matrix for Fold 8

Actual / Predicted	Undefined	Independent
Undefined	419	0
Independent	1	414

Source: Prepared by the authors, 2026.

From a quantitative perspective, Fold 8 achieved 414 true positives (TP), 419 true negatives (TN), 1 false negative (FN), and zero false positives (FP). Table 24 summarizes the classification metrics. The model attained a recall of 0.9976, perfect precision of 1.0000, and an F1-score of 0.9988, highlighting its robustness in minimizing critical classification errors.

Table 24 – Classification report for Fold 8

Metric	Precision	Recall	F1-score
Independent Class	1.0000	0.9976	0.9988

Source: Prepared by the authors, 2026.

Overall, the analysis of both selected models demonstrates that the proposed approach is capable of producing highly accurate and reliable classifiers under different optimization criteria. While Fold 7 excels in terms of overall generalization as indicated by the lowest validation error, Fold 8 provides superior performance in minimizing critical errors, offering a complementary perspective for model selection depending on application-specific requirements.

3.5.7 ANALYSIS OF THE WORST-PERFORMING MODELS

Although the proposed approach demonstrated highly stable performance across all folds, an analysis of the worst-performing cases provides additional insight into the lower bounds of the method under different selection criteria. Using the same criteria adopted for the best-performing models — highest validation error and highest critical error (False Positive Rate) — this section discusses the least favorable scenarios observed during the cross-validation process.

3.5.7.1 WORST MODEL ACCORDING TO THE HIGHEST VALIDATION ERROR

With respect to validation loss, **Fold 8** presented the highest validation error among all evaluated folds. This behavior indicates comparatively weaker generalization performance according to the Binary Cross-Entropy objective.

It is important to note that Fold 8 was previously analyzed in detail in **Section 3.5.6.2**, where it was identified as the best-performing model under the critical error criterion. Therefore, its training dynamics, confusion matrix, and classification report are not repeated here to avoid redundancy.

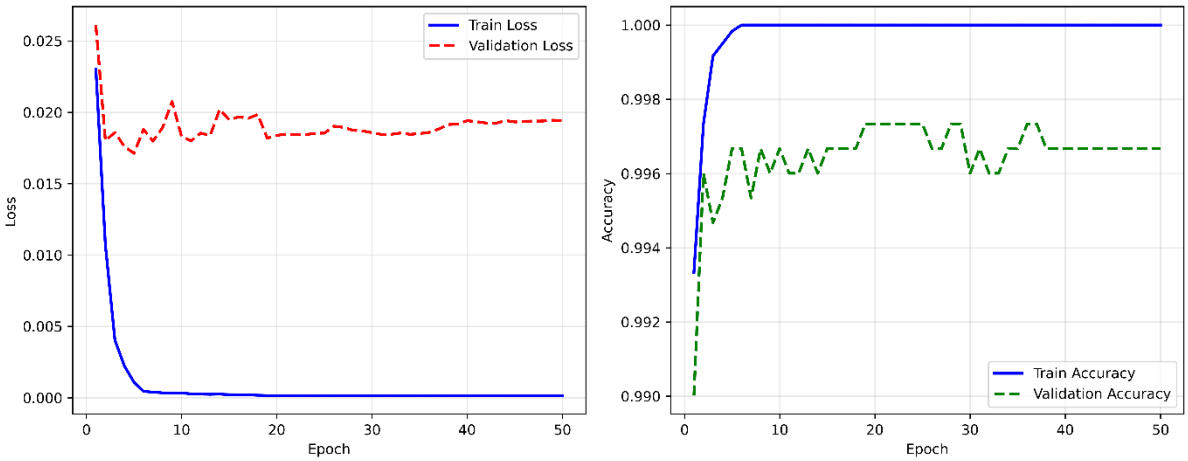
From a quantitative perspective, Fold 8 achieved 414 true positives (TP), 419 true negatives (TN), 1 false negative (FN), and no false positives (FP). Despite exhibiting the highest validation loss, the model maintains excellent classification performance, with a recall of 0.9976, perfect precision of 1.0000, and an F1-score of 0.9988. These results highlight a clear trade-off between optimization objectives, where minimizing validation error and minimizing critical errors may lead to different model selections.

3.5.7.2 WORST MODEL ACCORDING TO THE HIGHEST CRITICAL ERROR

Considering the False Positive Rate as the critical error criterion, **Fold 2** was identified as the worst-performing model, presenting the highest number of false positives among all folds. Although the absolute number of critical errors remains limited, this fold represents the least favorable case in scenarios where false alarms must be strictly minimized.

Figure 24 illustrates the training and validation loss and accuracy curves for Fold 2. The training loss rapidly converges toward zero, while the validation loss stabilizes without signs of divergence. Both training and validation accuracy remain high throughout the training process, indicating stable learning behavior despite the increased critical error.

Figure 24 – Training and validation loss and accuracy curves for Fold 2 (worst critical error)



Source: Prepared by the authors, 2026.

Table 25 presents the confusion matrix for Fold 2. The presence of three false positives directly explains its selection as the worst case under the critical error criterion.

Table 25 – Confusion matrix for Fold 2 (worst critical error)

Actual / Predicted	Undefined	Independent
Undefined	432	3
Independent	5	394

Source: Prepared by the authors, 2026.

Table 26 summarizes the classification metrics for Fold 2. While precision, recall, and F1-score remain above 0.98, this fold exhibits the highest False Positive Rate among the evaluated models, characterizing it as the least favorable case under the critical error perspective.

Table 26 – Classification report for Fold 2 (worst critical error)

Metric	Precision	Recall	F1-score
Independent	0.9924	0.9875	0.9899
Undefined	0.9886	0.9931	0.9908

Source: Prepared by the authors, 2026.

Overall, this analysis confirms that even the worst-performing models maintain high predictive performance. The observed variations mainly reflect trade-offs between optimization criteria rather than instability or poor generalization, reinforcing the robustness of the proposed approach.

3.6 DISCUSSION

The experimental results demonstrate that the proposed DistilBERT-based approach achieves consistently high performance in the task of classifying parallelization potential in source code. Across the 10-fold cross-validation, the model exhibited high accuracy on training, validation, and test sets, with narrow confidence intervals and low variability. The proximity between validation and test accuracies indicates that the model generalizes well to unseen data, with no evidence of overfitting, despite the high capacity of the Transformer architecture.

Beyond overall accuracy, the analysis of precision, recall, and F1-score confirms that the model maintains a balanced classification behavior. Precision remains consistently high across folds, indicating strong control over false positives, while recall values also remain above 99% on average, demonstrating effective identification of parallelizable loops. The resulting F1-scores reflect this balance, reinforcing the robustness of the classifier and its suitability for tasks where both correct detection and error minimization are critical.

The validation error analysis further supports the stability of the training process. The Binary Cross-Entropy loss exhibits low absolute values and a compact distribution across folds, confirming that the optimization procedure converges reliably. However, the comparison between validation error and application-critical metrics reveals an important trade-off: the model with the lowest validation loss is not necessarily the one that minimizes false positives. This finding highlights the relevance of considering multiple evaluation criteria when selecting models for practical deployment.

The False Positive Rate analysis provides additional insight into the model's reliability from an application-oriented perspective. The very low average FPR observed across all folds indicates that the model rarely misclassifies undefined loops as parallelizable. Notably, one fold achieved zero false positives, demonstrating that the proposed approach is capable of completely eliminating this critical error under certain conditions. This behavior is particularly relevant in scenarios where incorrect parallelization decisions may lead to unsafe or inefficient execution.

Finally, the examination of best- and worst-performing cases shows that performance variations are limited and primarily reflect trade-offs between optimization objectives rather than instability. Even the least favorable models maintain high accuracy and F1-scores, confirming the robustness of the approach. Overall, the results indicate that the proposed method effectively leverages Transformer-based representations for source code analysis, providing accurate, stable, and reliable classification within the defined scope of the study.

3.7 CONCLUSION

This work presented a Transformer-based approach for the automatic classification of parallelization potential in source code, focusing on the identification of independent loops that can be safely parallelized. The proposed methodology

combines a carefully constructed and balanced dataset—composed of synthetic codes generated via evolutionary algorithms and manually annotated real-world examples—with a fine-tuned DistilBERT model capable of processing source code directly as text. The experimental evaluation was conducted using a rigorous 10-fold cross-validation protocol and a comprehensive set of performance metrics.

The results demonstrate that the proposed approach achieves consistently high performance across all folds, with strong generalization capability and low variability. High accuracy, precision, recall, and F1-score values were observed on the test sets, indicating that the model effectively distinguishes between parallelizable and undefined loop structures. The statistical analysis, including confidence intervals and distribution visualizations, further confirms the stability and robustness of the learned models.

Beyond overall performance, the study highlighted the importance of considering application-specific evaluation criteria. While validation error proved effective for identifying models with strong generalization behavior, the analysis of the False Positive Rate revealed complementary insights, particularly for scenarios where incorrect classification of non-parallelizable code as parallelizable may lead to undesirable consequences. The ability of the proposed approach to achieve extremely low—and in some cases zero—false positives reinforces its suitability for practical code analysis contexts.

Despite these promising results, some limitations should be acknowledged. The scope of this study is restricted to loop-level analysis and does not account for more advanced loop transformations, such as fusion, unrolling, or tiling. In addition, although the inclusion of real-world code samples enhances representativeness, the dataset remains limited to a specific set of programming patterns and languages.

Future work will focus on extending the analysis to more complex code structures and transformations, expanding the dataset with additional real-world examples, and exploring the integration of the proposed classifier into static analysis tools or development environments. Investigating alternative Transformer architectures and multilingual code representations also represents a promising direction for further research.

Overall, the results indicate that Transformer-based models constitute a viable and effective solution for automatic parallelization potential classification, offering a

robust and scalable foundation for supporting parallel programming analysis within the defined scope of this study.

4 FINAL CONSIDERATIONS

This dissertation investigated the use of Artificial Intelligence techniques to support the automatic identification of parallelization opportunities in sequential source code, motivated by practical limitations observed in the Parallel Experience for Sequential Code (PESC) framework. Although PESC enables the exploitation of distributed infrastructures and idle computational resources, its workflow still depends on manual analysis to determine where parallel execution can be safely applied. Reducing this dependency was the central research challenge addressed throughout the studies presented here.

The work was developed as a sequence of two complementary investigations, each designed to increase realism, representational richness, and applicability while preserving methodological rigor.

In the first article, we examined whether relatively lightweight neural architectures could learn discriminative patterns associated with loop independence in a controlled environment. By adopting token-based representations and synthetically generated programs with explicit dependency constraints, the study isolated the core learning problem and minimized external sources of variability. The results confirmed the main hypothesis: even simplified models are capable of capturing relevant structural cues. In the best configuration, the CNN model achieved 97.67% test accuracy, with F1-scores reaching 0.98 for dependent loops and 0.97 for independent loops. These outcomes established a solid proof of concept and demonstrated that data-driven strategies are viable for assisting automatic parallelization.

The second article extended this foundation by incorporating Transformer-based representations through DistilBERT and by expanding the dataset to include both synthetic and real code. This transition introduced greater heterogeneity and brought the experiments closer to practical development scenarios. Even under these more demanding conditions, the approach maintained outstanding performance, reaching a mean test accuracy of 99.60% across the cross-validation folds. Most importantly, the model produced an extremely low false positive rate, with a mean value of 0.0029, reinforcing the reliability required in real execution environments

where incorrect parallelization decisions may compromise correctness. The findings therefore confirmed not only feasibility but also robustness.

From a broader perspective, the contributions of this dissertation can be summarized along three main dimensions. First, it provides empirical evidence that machine learning models can assist in recognizing parallel execution opportunities without relying exclusively on traditional compiler analyses. Second, it offers reproducible strategies for dataset construction, labeling, and evaluation in a domain where public benchmarks remain scarce. Third, it establishes a technological path toward integrating intelligent classifiers into platforms such as PESC, enabling higher levels of automation and scalability.

Naturally, some limitations remain. While the inclusion of real programs in the second study represents an important advance, further expansion of datasets, programming languages, and dependency patterns is necessary to fully characterize generalization capabilities. Moreover, integration with production-level parallelization pipelines demands additional validation layers, including static and dynamic safety checks.

As future work, promising directions include the exploration of larger language models, hybrid approaches that combine learning with formal verification, and continuous learning strategies capable of adapting to new coding styles and paradigms. Advancing along these lines may contribute to transforming automatic parallelization into a practical and widely accessible resource for software developers.

In conclusion, the results achieved throughout this dissertation indicate that Artificial Intelligence constitutes a consistent and effective avenue for mitigating one of the main bottlenecks in distributed execution frameworks. By progressively moving from controlled validation to realistic experimentation, the research demonstrates that the automatic identification of parallelizable regions is not only possible, but also increasingly reliable, opening the door to deeper integration between modern machine learning and high-performance computing environments.

REFERENCES

ALLAMANIS, M. *et al*. A Convolutional Attention Network for Extreme Summarization of Source Code. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING (ICML), 2016. **Proceedings [...]**. [S. l.: s. n.], 2016.

ALLAMANIS, M. *et al*. A Survey of Machine Learning for Big Code and Naturalness. **ACM Computing Surveys**, v. 51, n. 4, p. 1-37, 2018. Available at: https://doi.org/10.1145/3212695.

KOLMOGOROV, A. N. Sulla determinazione empirica di uma legge di distribuzione. **Giornale dell'Istituto Italiano degli Attuari**, v. 4, p. 89-91, 1933.

CHEN, L. *et al*. Learning to parallelize with OpenMP by augmented heterogeneous AST representation. In: PROCEEDINGS OF MACHINE LEARNING AND SYSTEMS (MLSYS), 4., 2022. **Proceedings [...]**. [S. l.: s. n.], 2022. p. 442-456.

CORREIA, I. S. *et al*. Discovering Software Parallelization Points Using Deep Neural Networks. **arXiv preprint**, 2025. Available at: https://arxiv.org/abs/2509.16215. Submitted for publication.

FENG, Z. *et al*. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. **arXiv preprint**, 2020. Available at: https://arxiv.org/abs/2002.08155.

FORTIN, F.-A. *et al*. DEAP: Evolutionary Algorithms Made Easy. **Journal of Machine Learning Research**, v. 13, p. 2171-2175, 2012.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambridge, MA, USA: MIT Press, 2016.

GOSSET, W. S. The probable error of a mean. **Biometrika**, v. 6, n. 1, p. 1-25, 1908. Available at: https://doi.org/10.1093/biomet/6.1.1.

GRAY, V. **Principal Component Analysis**: Methods, Applications, and Technology. New York, NY, USA: Nova Science Publishers, 2017.

GUO, D. *et al*. GraphCodeBERT: Pre-training Code Representations with Data flow. **arXiv preprint**, 2020. Available at: https://arxiv.org/abs/2009.08366.

HAREL, R. *et al*. Pragformer: data-oriented parallel source code classification with transformers. **International Journal of Parallel Programming**, v. 53, n. 1, p. 2-18, 2025. Available at: https://doi.org/10.1007/s10766-024-00416-1.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. Ann Arbor: University of Michigan Press, 1975.

KELLY, W.; PUGH, W. **The omega calculator and the omega library**. Maryland: University of Maryland, 1996. (Technical Report).

KINGMA, D. P.; BA, J. **Adam**: a method for stochastic optimization. [S. l.]: arXiv, 2014. Available at: https://arxiv.org/abs/1412.6980.

LIU, W. *et al*. Hierarchical attention networks for code smell detection. **Information and Software Technology**, v. 106, p. 107-119, 2019.

MAHMUD, Q. *et al*. AutoParLLM: GNN-guided automatic code parallelization using large language models. **International Journal of Parallel Programming**, 2025. In press.

PEARSON, K. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, v. 2, n. 11, p. 559-572, 1901.

PEDREGOSA, F. *et al*. Scikit-learn: Machine Learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825-2830, 2011.

POUCHET, L.-N. *et al*. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 1, p. 137-150, 2011.

SANH, V. *et al*. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. In: NEURIPS WORKSHOP ON ENERGY EFFICIENT MACHINE LEARNING AND COGNITIVE COMPUTING, 2019. **Proceedings [...]**. Vancouver, Canada: [s. n.], 2019.

SANTOS, H. C. T. *et al*. Pesc - parallel experience for sequential code. **Concurrency and Computation: Practice and Experience**, v. 37, n. 12-14, e70102, 2025.

SHARMA, S.; SPINELLIS, D. Deep learning-based detection of code smells: a systematic literature review. **Journal of Systems and Software**, v. 176, 110932, 2021.

SHEN, Y. *et al*. Towards parallelism detection of sequential programs with graph neural networks. **Future Generation Computer Systems**, v. 125, p. 515-525, 2021.

SMIRNOV, N. Table for estimating the goodness of fit of empirical distributions. **The Annals of Mathematical Statistics**, v. 19, n. 2, p. 279-281, 1948.

VASWANI, A. *et al*. Attention Is All You Need. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS (NEURIPS), 30., 2017. **Proceedings [...]**. [S. l.: s. n.], 2017.

WHITE, M. *et al*. Deep Learning Code Fragments for Code Clone Detection. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), 31., 2016. **Proceedings [...]**. [S. l.]: IEEE/ACM, 2016.

WOLF, T. *et al*. Transformers: State-of-the-Art Natural Language Processing. In: CONFERENCE ON EMPIRICAL METHODS IN NATURAL LANGUAGE

PROCESSING (EMNLP), 2020. **Proceedings [...]**. Online: Association for Computational Linguistics, 2020. p. 38-45.

ZINENKO, O. Dependence Analysis. **Encyclopedia of Parallel Computing**, [S. l.]: Springer, 2018.

ZINENKO, O. *et al*. Modeling the polyhedral representation of programs in the scheduling context. In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION (CC), 27., 2018. **Proceedings [...]**. [S. l.]: ACM, 2018. p. 1-12.