University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Integrating Fine-Tuned LLMs and LSTM-Based Knowledge Tracing for Adaptive Learning in C Programming: A Neo-Piagetian Approach

Supervisor
Prof. Giovanni Da San Martino
University of Padova

Master Candidate
Michelle Elizabeth Sánchez Puma

Student ID
2141808

Academic Year
2025-2026

Abstract

Students in an introductory C programming course frequently struggle with code understanding because achieving proficiency requires more than syntactic memorization; it demands a gradual understanding of abstract code concepts. There exist specific learning models like the Neo-Piaget framework, according to which novice programmers must base their learning on a progression of distinct cognitive stages to develop a true programming understanding. This framework guides student development across three stages of increasing complexity: sensorimotor, pre-operational, and concrete operational stages. Guiding students through these stages typically requires personalized one-to-one interaction, which is often unfeasible in large educational settings. To address this challenge, Large Language Models (LLMs) are an artificial intelligence (AI) system that could, in principle, simulate individual tutoring by using their capabilities in tasks like code generation, debugging, and natural language explanation. However, the learning by stages proposed by the Neo-Piaget theory is not covered by traditional AI tools, which are not oriented to identify specific cognitive barriers that prevent the correct abstraction of student learning.

This thesis proposes an integrated architecture that leverages LLMs and knowledge tracing (KT) to provide a precise cognitive diagnosis, adaptive remediation, and academic risk prediction. The methodology involves the sequential use of three components. First, a *Diagnostic Engine* driven by a fine-tuned open-weight LLM that identifies missing programming skills and classifies student understanding within the Neo-Piagetian stages directly from exercise code submissions. Second, a *Remediation Engine* utilizes few-shot prompting techniques to generate tailored, appropriate exercises based on the student's actual cognitive level. Finally, a *Predictive Engine* uses these enriched diagnostics as inputs for a neural network to forecast the probability of student exam failure. These three components are then integrated into a functional web application designed to provide actionable feedback to both instructors and learners.

The results demonstrate that the fine-tuned open-weights model achieved a diagnostic accuracy of 74% in cognitive-stage classification, outperforming general-purpose commercial LLMs. Furthermore, by forcing adherence to the learning framework through prompt engineering, the system successfully generated stage-appropriate exercises. Consequently, integrating these precise cognitive labels into the knowledge tracing system improved the prediction of student outcomes for the first partial exam from a baseline of 67% to 76%. Finally, the proposed system provides a solution for personalized C programming learning based on a pedagogical framework, with the goal of future real-world classroom deployments to validate its impact on final exam results.

Contents

Abstract v

List of figures ix

List of tables xiii

Listing of acronyms xvii

1 Introduction 1

2 Background 4
- 2.1 Related Work . . . 4
- 2.2 Learning Framework . . . 7
  - 2.2.1 Neo-Piaget Cognitive Stages for Programming Learning . . . 7
  - 2.2.2 The Block Model for Programming Comprehension . . . 9
- 2.3 Machine Learning Technologies . . . 10
  - 2.3.1 Large Language Models and Prompt Engineering . . . 11
  - 2.3.2 Fine-Tuning and Knowledge Distillation . . . 11
  - 2.3.3 Knowledge Tracing and LSTM Networks . . . 12

3 Proposed Framework and System Architecture 13
- 3.1 Integrated Learning Framework . . . 14
- 3.2 Exercise Typologies Derived from the Integrated Learning Framework . . . 15
- 3.3 Topics and Skills . . . 17
- 3.4 System Architecture . . . 22

4 The Diagnostic Engine: Iterative Design and Evaluation 27
- 4.1 Quantitative Evaluation Strategy . . . 28
- 4.2 Baseline Prompt for Missing Skills Diagnostic . . . 35
  - 4.2.1 Methodology Design and Implementation . . . 35
  - 4.2.2 Baseline Results . . . 37
- 4.3 Improving Baseline Prompt through Enhanced Prompt Engineering . . . 53
  - 4.3.1 Methodology Design and Implementation . . . 53
  - 4.3.2 Enhanced Prompt Results . . . 55
- 4.4 Using Fine-Tuning for Improving Model Accuracy . . . 62
  - 4.4.1 Dataset Construction & Knowledge Distillation . . . 63

4.4.2 Baseline Fine-Tuning Architecture 65
4.4.3 Fine-Tuning Architecture using Chain of Thought (CoT) 70
4.4.4 Fine-Tuning Attempt Using More Aggressive Training Parameters . . 74
4.4.5 Fine-Tuning Attempt Using an Augmented Dataset 78

5 The Remediation Engine: Iterative Design and Evaluation 89
5.1 Qualitative and Quantitative Evaluation Strategy 90
5.2 Baseline for Evaluating LLM Alignment with the Learning Framework . . . 91
5.2.1 Methodology Design and Implementation 91
5.2.2 Baseline Results 92
5.3 Evaluating the LLMs Ability to Generate Complex and Varied Exercises . . . 95
5.3.1 Methodology Design and Implementation 96
5.3.2 LLMs Capabilities Results 97
5.4 Evaluating LLM Adoption of Exercise Typology Guidelines 100
5.4.1 Methodology Design and Implementation 100
5.4.2 LLM Adoption for Exercise Typology Results 102
5.5 Few-Shot Prompting for Evaluating LLM Adoption of Exercise Typology . . 105
5.5.1 Methodology Design and Implementation 105
5.5.2 Few-Shot Prompting Results 108

6 The Predictive Engine: Iterative Design and Evaluation 112
6.1 Quantitative Evaluation Strategy 113
6.2 Dataset Construction and Feature Engineering 114
6.2.1 Dataset Construction 114
6.2.2 Dataset Encoding 116
6.3 Knowledge Tracing Training 117
6.3.1 Training using Large Dimensionality Encoding: 118
6.3.2 Training using Normalized Scores Encoding: 118
6.3.3 Training using Bi-LSTM with Attention: 123

7 Web Application Pilot Deployment 131

8 Discussion 135
8.1 The *Diagnostic Engine*: The Feasibility of Cognitive Diagnosis 135
8.2 The *Remediation Engine*: LLMs and Learning Framework Constraints . . . 137
8.3 The *Predictive Engine*: Improvements in Knowledge Tracing 138
8.4 Pilot Deployment Viability and Future Work Directions 139

9 Conclusions 141

References 143

Listing of figures

3.1 General System Workflow for Student interactions. 24
3.2 Proposed System Architecture 25
3.3 Student Dashboard of the proposed System. 26
3.4 Professor Dashboard of the proposed System. 26

4.1 Generate Exercises Prompt for Ground-Truth dataset. 32
4.2 Generate Answers Prompt for Ground-Truth dataset. 33
4.3 Baseline Prompt for Cognitive Framework Diagnostic. 37
4.4 Accuracy for Baseline Prompt in Gemini 2.5-Flash Model 41
4.5 Confusion Matrix for Baseline prompt in Gemini 2.5 Flash Model 41
4.6 Accuracy and F1-score by Missing Skill for Baseline Prompt in Gemini 2.5 Flash Model. 43
4.7 Accuracy and F1-score by Mastered Skill for Baseline Prompt in Gemini 2.5 Flash Model. 43
4.8 Stage Distribution for Baseline Prompt in Gemini 2.5 - Flash Model. 44
4.9 Accuracy for Baseline Prompt in GPT-OSS-120b Model 45
4.10 Confusion Matrix for Baseline Prompt in GPT-OSS-120b Model 45
4.11 Accuracy by Missing Skill for Baseline Prompt in GPT-OSS-120b Model . . . 47
4.12 Accuracy by Mastered Skill for Baseline Prompt in GPT-OSS-120b Model . . 47
4.13 Stage Distribution of the Large-Scale Evaluation for GPT-0SS-120b Model. . 48
4.14 Accuracy for Baseline Prompt in GPT-OSS-20b Model 49
4.15 Confusion Matrix for Large-Scale Evaluation in GPT-OSS-20b Model 49
4.16 Accuracy by Missing Skill for Baseline Prompt in GPT-OSS-20b Model . . . 51
4.17 Accuracy by Mastered Skill for Baseline Prompt in GPT-OSS-20b Model . . . 51
4.18 Stage Distribution for Baseline Prompt in GPT-0SS-20b Model. 52
4.19 Chain of Thought Strategy Prompt for Cognitive Framework Diagnostic. . . 55
4.20 Confusion Matrix for Enhanced Prompt in Gemini 2.5-Flash model. 57
4.21 Confusion Matrix for Enhanced Prompt in GPT-OSS-120b model. 59
4.22 Confusion Matrix for Enhanced in GPT-OSS-20b model. 61
4.23 Fine-Tuning Process for the *Diagnostic Engine*. 63
4.24 Loss Curve for Fine-Tuning process. 69
4.25 Confusion Matrix in Fine-Tuned GPT-OSS-20b model. 71
4.26 Loss Curve for Fine-Tuning process using CoT. 72
4.27 Confusion Matrix in Fine-Tuned GPT-OSS-20b model with CoT. 74

4.28 Loss Curve for Fine-Tuning process using CoT and a more aggressive parameter configuration. 75
4.29 Confusion Matrix in Fine-Tuned GPT-OSS-20b model with CoT and a more aggressive parameter configuration. 77
4.30 Reasoning step of the Fine-tuned model using CoT training. 78
4.31 Prompt for generating new code submissions for the Fine-tuned augmented dataset. 82
4.32 Loss Curve for Fine-Tuning process using the augmented dataset. 84
4.33 Confusion Matrix in Fine-Tuned GPT-OSS-20b model using the augmented dataset. 86
4.34 Heatmap of Missing Skills for Large-Scale Evaluation in Fine-Tuned GPT-OSS-20b model using the augmented dataset. 86
4.35 Heatmap of Mastered Skills for Large-Scale Evaluation in Fine-Tuned GPT-OSS-20b model using the augmented dataset. 87

5.1 Prompt for Evaluating LLM General Alignment with Learning Framework. . 92
5.2 Prompt for Evaluating LLMs Ability to Generate Complex Exercises. 97
5.3 Prompt for Evaluating LLM Adoption of Exercise Typology Guidelines. . . . 102
5.4 Prompt for Evaluating LLM Adoption of Exercise Typology using Few-shot strategies. 108

6.1 Loss Curves of 5-Folds for the Knowledge Tracing Model using Large Dimensionality Encoding. 119
6.2 Confusion Matrix for the Knowledge Tracing Model using LSTM with Attention architecture. 120
6.3 ROC Curve for the Knowledge Tracing Model using LSTM with Attention architecture. 121
6.4 Loss Curves of 5-Folds for the Knowledge Tracing Model using LSTM with Attention architecture. 121
6.5 Attention Map - False Positive for the Knowledge Tracing Model using LSTM with Attention architecture. 122
6.6 Attention Map - True Negative for the Knowledge Tracing Model using LSTM with Attention architecture. 123
6.7 Confusion Matrix for the Knowledge Tracing Model using Bi-LSTM with Attention architecture. 124
6.8 ROC Curve for the Knowledge Tracing Model using Bi-LSTM with Attention architecture. 125
6.9 Loss Curves of 5-Folds for the Knowledge Tracing Model using Bi-LSTM with Attention architecture. 125
6.10 Attention Map - True Positive for the Knowledge Tracing Model using Bi-LSTM with Attention architecture. 126

6.11 Attention Map - True Negative for the Knowledge Tracing Model using Bi-LSTM with Attention architecture. 127
6.12 ROC Curve for the Knowledge Tracing Model using GRU with Attention architecture. 127
6.13 Confusion Matrix for the Knowledge Tracing Model using baseline without enhanced labels. 128
6.14 ROC Curve for the Knowledge Tracing Model using baseline without enhanced labels. 129

7.1 Final Professor Dashboard in the pilot deployment. 132
7.2 Successful Learning Interactions of a student in the Logical Expressions topic. 133
7.3 Inconvenient in Learning Interactions of a student in the Expressions topic. . 134

Listing of tables

2.1 Neo-Piagetian cognitive stages. 9
2.2 Block Model Representation, taken from Schulte et al. [1]. 10

3.1 Mapping between Neo-Piagetian stages and the Block Model. 15
3.2 Exercise Types Mapped to Neo-Piagetian Cognitive Stages 16
3.3 List of topics and skills, mapped to Neo-Piaget cognitive stages. 17

4.1 Exercises obtained by Moodle Log Analysis 29
4.2 Ground-truth Datasets for Large-Scale Evaluation 29
4.3 Exercises generated by Claude Sonnet 4.5 30
4.4 Final number of Exercises for the Ground-Truth dataset 34
4.5 Strengths and Weaknesses of the Models in Identifying Missing Skills Test . . 40
4.6 Quantitative Evaluation for Gemini 2.5 Flash Model - Metrics by Cognitive Stage 40
4.7 Quantitative Evaluation for Gemini 2.5 Flash Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills 42
4.8 Quantitative Evaluation for GPT-OSS-120b Model - Metrics by Cognitive Stage 44
4.9 Quantitative Evaluation for GPT-OSS-120b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills 46
4.10 Quantitative Evaluation for GPT-OSS-20b Model - Metrics by Cognitive Stage 48
4.11 Quantitative Evaluation for GPT-OSS-20b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills 50
4.12 Accuracy and F1-score for all models in the Large-Scale Evaluation Prompt . . 52
4.13 Quantitative Evaluation for Gemini 2.5-Flash Model using the enhanced prompt - Metrics by Topic 56
4.14 Quantitative Evaluation for Gemini 2.5-Flash Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) 56
4.15 Quantitative Evaluation for GPT-OSS-120b Model using the enhanced prompt - Metrics by Topic 58
4.16 Quantitative Evaluation for GPT-OSS-120b Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) 58
4.17 Quantitative Evaluation for GPT-OSS-20b Model using the enhanced prompt - Metrics by Topic 60
4.18 Quantitative Evaluation for GPT-OSS-20b Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) 60

4.19 Accuracy and F1-score for all models in the Large-Scale Evaluation for the Enhanced Prompt . . . 62
4.20 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model - Metrics by Topic . . . 69
4.21 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) . . . 70
4.22 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT - Metrics by Topic . . . 72
4.23 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) . . . 73
4.24 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT and a more aggressive parameter configuration - Metrics by Topic . . . 75
4.25 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT and a more aggressive parameter configuration - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) . . . 76
4.26 Total Number of Submissions by Cognitive Stage of the dataset for the Fine-tuning process. . . . 79
4.27 Number of Submissions by Cognitive Stage for each topic of the dataset for the Fine-tuning process. . . . 79
4.28 Types of errors observed in generated code submissions for the augmented dataset during the fine-tuning process. . . . 80
4.29 Total Number of Submissions by Cognitive Stage of the Augmented dataset for the Fine-tuning process. . . . 83
4.30 Number of submissions by cognitive stage for each topic in the augmented dataset used for the fine-tuning process. . . . 83
4.31 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model using the augmented dataset - Metrics by Topic . . . 84
4.32 Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model using the augmented dataset - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills) . . . 85
4.33 Accuracy and F1-score comparison for all models after the fine-tuning process. 88

5.1 Comparison of LLMs in Prompt 1 (single-column width) . . . 95
5.2 Exercise Diversity per Model in Test 2 Prompt for Exercise Generation Task . 99
5.3 Exercise Alignment with Typology Guidelines in Test 3 Prompt for Exercises Generation . . . 104
5.4 Exercise Alignment with Typology Guidelines in Test 4 Prompt for Exercises Generation . . . 110

6.1 First Experiment of Encoding for the Dataset for the Knowledge Tracing Process. Variables are abbreviated for formatting constraints (e.g., C_S1 = COND_S1, Sens. = isSensorimotor). . . . 117

6.2 Second Experiment of Encoding for the dataset for the Knowledge Tracing Process. Variables are abbreviated to fit formatting constraints (e.g., EID = EXERCISE_ID, LEXS = LOGEX_SCORE, STG = STAGE). 117
6.3 Performance comparison of Deep Knowledge Tracing using BiLSTM with Attention, with and without enriched labels. 129

Listing of acronyms

AI . . . . . Artificial Intelligence

LLM . . . . . Large Language Models

RAG . . . . . Retrieval Augmented Generation

API . . . . . Application Programming Interface

GPT . . . . . Generative Pretrained Transformers

GPT-OSS . . . . . Generative Pretrained Transformers - Open Source Software

vLLM . . . . . Virtual Large Language Model

LoRA . . . . . Low-Rank Adaptation

PEFT . . . . . Parameter-efficient Fine-tuning

RNN . . . . . Recurrent Neural Networks

LSTM . . . . . Long-Short Term Memory

Bi-LSTM . . . . . Bidirectional Long-Short Term Memory

Acknowledgments

I thank God for His continuous blessings and guidance throughout my life, enabling me to successfully achieve my goals.

To my family, for their love, sacrifices, and unconditional support, and for always encouraging me to pursue my dreams.

To my thesis advisor, Prof. Giovanni Da San Martino, for his invaluable guidance, help, and encouragement, which were instrumental in the successful completion of this project.

To the University of Padua and all my professors, for the knowledge they imparted, profoundly shaping both my academic and professional development.

1 Introduction

Learning programming is one of the foundational pillars in computer science careers, and introductory programming courses are part of all bachelor degrees. Although various languages are used to introduce it, the C language is known for requiring students to understand the use of low-level processes, such as memory allocation and array manipulation, making it a robust language for a deep understanding of how software interacts with computer architecture. Consequently, the complexity of learning C involves many difficulties for a novice programmers [2] due to the high level of logical abstraction that it requires.

For that reason, learning programming is inherently a cognitive developmental process because students have to understand increasingly complex coding topics. To master them, students are forced to go through distinct stages to acquire skills at different levels. Several learning models have been proposed in the literature for learning programming, such as the Neo-Piaget framework, which suggests that learning is achieved by stages of increasing complexity [3]. According to this framework, novice programmers must overcome different cognitive stages before they can successfully understand advanced and abstract programming concepts. These stages guide their cognitive development from the sensorimotor stage, where code is perceived as static text; to the pre-operational stage, where they can manually trace execution but lack abstraction; and finally to the concrete operational stage, where they can understand and abstract the code purpose [3]. In that way, learners do not stay only in syntax memorization; instead, they are able to achieve deep conceptual understanding.

Because students progress through these cognitive stages at different paces, a standard class-

room approach in which every student follows the same path is often insufficient. Overcoming individual cognitive barriers requires highly personalized one-to-one tutoring to guide each student through their specific stage, which is often unfeasible in large courses [4]. In order to address this limitation, recently Large Language Models (LLMs) tools offer opportunities as personalized tutors for tasks like code generation or natural language explanation. However, their default behavior often acts as problem solvers by providing students with a complete code solution rather than guiding them through explanations for true programming comprehension [5]. With the aim of overcoming this gap, this research proposes a tutoring system that integrates the generative potential of LLMs and Knowledge Tracing (KT) with the constraints of the Neo-Piagetian learning framework to create a personalized system that helps students individually advance beyond each cognitive stage.

For that reason, the main objective of this thesis is to design and implement an autonomous tutoring system capable of diagnosing cognitive stages, generating tailored exercises, and predicting the final academic risk. To achieve this, the research is guided by the following research questions.

- **RQ1 (Diagnostic):** How accurately can LLMs diagnose a student's cognitive stage and missing pre-defined programming skills, within the Neo-Piaget learning framework, directly from C code exercise submissions?
- **RQ2 (Remediation):** How effectively do LLMs adhere to the constraints of the Neo-Piaget learning framework when generating complex and varied C programming exercises?
- **RQ3 (Prediction):** How does the integration of LLM predicted information on Neo-Piaget stages into traditional knowledge tracing improve the prediction of academic risk in C programming courses?

To address these research questions, this project proposes a three-phase tutoring system consisting of a *Diagnostic Engine*, a *Remediation Engine*, and a *Predictive Engine*. Through this architecture, the research offers the following main contributions to the field of computer science education:

- **A Neo-Piagetian Skill Framework:** The definition of a list of introductory C programming topics, where each atomic skill is explicitly mapped to its corresponding Neo-Piaget cognitive stage.

- **A Manually Annotated Cognitive Dataset:** The creation of a dataset based on real code submissions extracted from Moodle logs of past programming courses. These historical submissions were manually annotated with specific missing skills and their corresponding Neo-Piagetian cognitive stages, establishing the ground-truth data required for the analysis process of the next engines.
- **A Diagnostic Engine:** The development of a diagnostic module that uses a fine-tuned open-weight Large Language Model. This engine is able of extracting pedagogical information like cognitive stages and missing skills directly from a student's coding exercise solution.
- **A Remediation Engine:** An instructional module that leverages few-shot prompting to dynamically generate personalized programming exercises, which are strictly adapted to the diagnosed cognitive stage of the student.
- **A Predictive Engine:** A Knowledge Tracing model that processes the sequential history of student interactions enriched with the Neo-Piaget cognitive labels to forecast the probability of academic failure in final programming examinations.
- **An Integrated Educational Platform:** The consolidation of the three components into a fully functional web application designed for practical deployment in real programming classes.

The thesis is structured as follows. Chapter 2 examines the theoretical background, explaining the Neo-Piaget learning framework and its application to this project, and the underlying machine learning technologies.Chapter 3 presents the proposed integrated educational framework together with the overall system architecture. Chapters 4, 5, and 6 describe the iterative design, methodological implementation, and experimental results for the three core components: the *Diagnostic Engine*, the *Remediation Engine*, and the *Predictive Engine*, respectively. Chapter 7 reports on the pilot deployment of the integrated web application. Chapter 8 explains the results obtained and includes information about the pilot deployment of the web application. Finally, Chapter 9 concludes and summarizes the main contributions, including directions for future work.

2 Background

2.1 Related Work

The integration of Generative AI, specifically Large Language Models (LLMs), into computer science education has been extensively explored in recent literature [6] [4] [7]. Based on their natural language generation capabilities, these models have evolved from simple auto complete tools into interactive educational tutors [8]. Their current applications range from automated syntax debugging and code generation to complete tutors who can solve different programming exercises. For that reason, researchers have been investigating the pedagogical potential and its limitations in the task of guiding students through complex challenges.

One of the most important applications are LLM agents, a combination that leverages long-term memory for foundational knowledge and short-term memory for real-time interactions between students and the model [7]. These agents can be divided into pedagogical agents and domain specific agents. Pedagogical agents automate learning recommendations for teachers and support students in code generation and error detection; and domain-specific agents which offer solutions to help students learn very specific topics [7]. However, the use of LLMs in education has a number of shortcomings, such as hallucinations that present inaccurate information as factual, or the critical foundational problem of LLMs: the impediment of students to generate in-depth learning due to their over-reliance on AI models [9].

A recent study presents an LLM-based chatbot as a platform for both students and profes-

sors in higher computer science education [10]. Their approach uses Retrieval Augmented Generation (RAG) to provide the necessary context to the LLM about the class materials, and aims to generate exercises and solve students questions. In general, this study uses the chatbot as a complementary material to traditional classes, demonstrating its efficacy with a 88% of accuracy of aligned responses. As reported in the study, students asked the model to generate the answer to some exercises and asked for more explanations about it, which could also imply a lack of deep learning understanding.

Another example of the use of AI as a programming tutor is AI-Lab [11], which is a framework to guide students in tasks such as topic introductions, detailed examples, and debugging assistance. This study is based on the concept that, by identifying and rectifying coding errors, students can enhance their learning process. The authors conclude with 54.5% of students using the framework for homework, but highlight the risks of creating an over-dependence on generative AI.

Despite these important technological advances, a significant part of current research uses LLMs primarily as problem solvers rather than as pedagogical agents. Although these models excel at correcting syntax or generating examples, they exhibit a limitation about the students dependence of getting direct code solutions instead of a guidance through the learning process without giving an explicit answer. For that reason, the lack of this tutor guiding constraint is a gap this thesis aims to overcome: to improve learning understanding through a correct guiding process based on cognitive barriers instead of just giving direct answers to students.

In order to mitigate the limitations of this unstructured AI tutoring, several researchers have attempted to include established educational paradigms in automated AI systems. For instance, Mudongo et al. [12] examined how the most common errors in java programming made by first-year students can be mapped into specific cognitive stages of a well-defined learning framework such as Piaget. The Piaget learning framework in general describes learning as different cognitive stages that people need to overcome to obtain a complete understanding of a specific topic [13]. The study proposed by Mudongo et al. [12] show how programming topics can be successfully adapted to the Piaget framework; however, it does not use any other method to include the use of generative AI within this framework.

Another example of the use of learning frameworks in programming education is described by Gluga et al. [14], who mention that a framework can be used as a way to classify materials used in learning and assessment, and proposes a web tutorial to categorize assessment tasks into cognitive labels. This study proves the feasibility of using cognitive stages as class materials to improve programming learning; however, the authors do not propose a solution to integrate

the learning framework for the materials of the students.

The authors in Zhang et al. [15] propose a model in which they use LLMs in order to correct student responses to programming questions based on the SOLO Taxonomy that aims to classify student responses into four stages. This study shows that GPT-4o achieved high accuracy in classifying responses in the simpler stages, but struggled to classify in the upper stages. Similarly, the study presented by Alidad et al. [16] use Bloom's taxonomy to generate exercises for novice programmers depending on each established stage, finding that, in general, LLMs can generate adequate exercises on each level; however, they do not present a classification of responses into the learning taxonomy; instead, they measure the alignment of LLMs into the specified levels.

Although these studies represent a critical step in trying to align Generative AI into learning frameworks, there remains a notable gap in the literature on generating a complete learning flow constraint to a rigid framework based on real student responses. In particular, few studies have analyzed the use of LLMs to improve programming learning based on the Neo-Piaget learning framework which establishes cognitive barriers to understand abstract concepts.

However, identifying these cognitive barriers in the students learning trajectory is insufficient to guaranty long-term academic success. In order to achieve the complete learning flow mentioned previously, an educational system should also track how a student evolves and overcomes these stages across multiple interactions. This time tracking of the student mastery of programming skills leads to the domain of Knowledge Tracing (KT). Within the field of forecasting student performance, the introductory topic of Knowledge Tracing based on deep neural networks was proposed by Piech et al.[17], who use recurrent neural networks to model the non-linear trajectories of student learning over time.

Some studies have been conducted on the use of knowledge tracking to predict academic performance. For instance, Amaya et al [18] present a state-of-the-art review on different machine learning models to implement the tracing of learning trajectories, concluding that the LSTM models have the best accuracy. Going further, Li et al.[19] propose the Teacher Thinking Knowledge Tracing model (2T-KT), which uses LLMs to enrich the input of the knowledge graph based on four elements: observation, guideline, interpretation and cognition. This study concludes that the 2T-KT model can achieve excellent student knowledge tracing performance, but it requires a larger amount of tokens to generate the augmented annotations. Although this study uses LLMs to provide more information to the model for the prediction task, these enriched labels are not based on a learning framework or taxonomy that defines their meaning. Similarly, Scarlatos et al.[20] use the complete interactions between students

and LLMs as part of the input to the knowledge tracking model to track student knowledge labels throughout the dialog, concluding that the use of this context information outperforms traditional binary predictive methods.

Even though the majority of existing Knowledge Tracing architectures still rely on traditional binary interactions, there exist some studies that try to use enriched labels based on LLMs demonstrating that this idea outperforms traditional methods. However, no one of them creates a structure label that encapsulates the student interactions as input for the model. For that reason, this project addresses this limitation by integrating rich labels generated by a fine-tuned LLM that is based on a well defined Neo-Piaget learning framework, thereby feeding the predictive model with the reasoning behind a failure of a student response.

2.2 Learning Framework

In order to use Large Language Models as programming tutors with a pedagogical background, their interactions must be strictly governed by a well-defined learning framework. This section outlines the theoretical foundation that guides the basis of the proposed tutoring system. First, it explores the Neo-Piagetian learning model, which focuses on categorizing novice programmers understanding into defined cognitive stages. Second, it studies the Block Model as a framework for classifying code comprehension on distinct structural elements. Then, this research proposes a novel integration of both theories to create a unified framework. Finally, this section details how this integrated framework is translated into exercise typologies and programming skills to be the basis for the learning framework that supports the AI models of the proposed project.

2.2.1 Neo-Piaget Cognitive Stages for Programming Learning

The foundational theory of cognitive development was originally established by the Swiss psychologist Jean Piaget, who proposed that human intelligence evolves through different stages; especially in child pedagogy due to biological maturation of the brain [21]. In contrast, the Neo-Piaget theory as mentioned by Lister et al. [13] is a variant of the Piaget theory that does not focus only on children; instead, it establishes that people regardless of their age progress through increasingly advanced stages of reasoning as they gain expertise in a specific problem domain. Based on these theories, multiple studies have been conducted in the field of computer science education, especially programming teaching, in which the focus is on understanding

how novice programmers can achieve abstract concepts.

For instance, Lister et al. [3] mention that programming teaching is based mainly on problem solving, and students who perform poorly in final examinations are due to a lack of skills prior to problem solving, such as code tracing. Furthermore, the author exposes previous experiments in which another variable emerges: the explain in words task, creating a direct relationship between tracing, explaining, and writing code. These are the premises on how the Neo-Piaget theory is built.

The Neo-Piaget learning framework used in this project is based on the three-stage model for mental development applied to programming learning [3] [13]. This model explains the way novice programmers learn by describing the cognitive stages as follows:

1. **Sensorimotor (Pre-Tracing):** Students in this stage have a sparse understanding about the code, without having enough expertise to trace the variables in the code. For that reason, the author considers that students can perceive code in a different way than programmers. This means that students basically understand the code as static text and not as executable code. According to the author, this is due to some reasons: first, students have common misconceptions about simple programming concepts, most of which are listed in Appendix A of the work presented by Sorva et al. [2]. Furthermore, they do not understand how computers work, and consequently, they believe that computers somehow understand what they want.

2. **Pre-operational (Tracing):** In this stage, students can reliable trace code but do not abstract from it. This means that students perceive the code as a set of variables that change their values across some code lines. For that reason, when asked about the purpose of the code, they generate a set of initial values, trace the behavior of those variables, and infer the objective of the code by comparing the initial and final values of the variables. Consequently, in this stage, students are not well positioned to write code independently. Finally, the author mentions that the only way to overcome this stage is to trace many code examples.

3. **Concrete Operational (Post-Tracing):** In this final stage, students can write well-designed code because they start to abstract the code and the diagrams they can define. As mentioned by the author, the main difference between Pre-operational and Concrete Operational stages is the way students trace the code, in this third stage, students do not initialize values for variables when they do tracing, instead they mentally use multiple possible values. In this stage, students can correctly answer the "explain in words" question in which instead of giving a line-by-line explanation of the code, they summarize the global objective of it.

As noted, Lister et al. [3] only present three stages, but the original Piaget and Neo-Piaget models propose four stages. In this case, the fourth stage is the Formal Operational Stage; however, this is classified as an expert in the domain, which cannot be considered as a novice programmer of an introductory course and for that reason is not included in this classification. Finally, table 2.1 presents a condensed summary with the main features of each of the Neo-Piaget stages.

Table 2.1: Neo-Piagetian cognitive stages.

Stage	Description
Sensorimotor	Learners read code as static text and do not trace its execution.
Pre-operational	Learners cannot write code independently but can trace it by assigning specific values to variables.
Concrete Operational	Learners are able to write code as they can form abstractions.

2.2.2 The Block Model for Programming Comprehension

The Neo-Piaget framework outlines the stages a learner is expected to overcome when learning programming skills. Based on this idea, a new approach called the Block Model is introduced, which focuses on clarifying how the structure of the code can be understood. The Block Model was first introduced by Schulte et al. [1] as a model specialized in learning programming based on reading code instead of writing it.

This model is structured as a matrix of 3 dimensions and 4 levels, in which each cell represents an aspect of the learning process [1], the table 2.2 shows the matrix organization.

As explained by Izu et al. [22], the vertical dimensions (A,B,R,M) go from simpler artifacts in a code, such as atoms that represent basic elements of the code. Then, blocks are sequences of related assignments; and relationships mean the relations between blocks. Finally, macrostructure level that represents the complete comprehension of the overall structure of the program text. In addition, the horizontal dimension of the model represents a hierarchy of increasingly complex programming structures. First, the text surface references the code as static text; then, the program execution level uses code as a dynamic entity; and finally, the function level, which represents the code as an overall purpose entity. Based on this representation, the block model can be read upward and rightward with an increase in complexity in each direction [1].

The purpose of using this educational framework was to define learning activities that can explicitly teach all components of programming comprehension [22]. For that reason, this

Table 2.2: Block Model Representation, taken from Schulte et al. [1].

Level	Text Surface (T)	Program Execution (P)	Function/Purpose (F)
(M) Macrostructure	Understanding the overall structure of the program text.	Understanding the *algorithm* underlying a program.	Understanding the goal/purpose of the program (in context).
(R) Relationships	Relations and references between blocks (e.g., method calls, object creation, data access).	Sequence of method calls, object sequence diagrams.	Understanding how subgoals relate to goals and how functionality is achieved.
(B) Blocks (Chunks)	*Regions of Interest* (ROI) that syntactically or semantically build a unit.	Operations of a block, method, or ROI.	Understanding the function of a block as a subgoal.
(A) Atoms	Language elements.	Operation of a statement.	Function of a statement: its purpose is understood only in context.
Architecture / Structure Dimensions		**Relevance / Intention Dimension**	

multidimensional matrix that categorizes how a programmer reads and understands software artifacts is a basic element in learning frameworks in computer science.

2.3 Machine Learning Technologies

The previous sections described the learning framework and the theoretical basis that the proposed system will use; in this section, the focus is on describing the specific machine learning algorithms and architectures used in the practical implementation of the system. Specifically, it introduces the generative capabilities of Large Language Models, the use of Knowledge Distillation to increase reasoning abilities, and the predictive power of Long Short-Term Memory (LSTM) networks. All of these models and where they were used are described in detail in the methodology section.

2.3.1 Large Language Models and Prompt Engineering

Large Language Models (LLMs) are a category of deep learning models capable of understanding and generating natural language. They are based on a Transformer architecture which allows them to analyze sequences of words and capture patterns in text. In essence, LLMs are trained on millions of data and work as statistical prediction machines that continuously predict the next word in a sequence to generate text [23].

In order for a LLM to generate text, it needs an initial interaction that acts as a clue of what a user needs; this is called a prompt. Prompt engineering is the practice to structure prompts to effectively guide generative AI models such as LLMs; for that reason, good prompts generate good responses [24]. It presents multiple techniques in order to improve prompts, one of them is Few-shot prompting. It is based on providing the AI with some examples to illustrate what the user final need is, so the LLM will learn from the examples by recognizing patterns that improve responses [24].

In this project, LLMs are extensively used with prompt engineering to force the models to follow the constraints of the learning framework; different techniques are applied to obtain the best responses according to the learning task.

2.3.2 Fine-Tuning and Knowledge Distillation

In general, commercial LLMS have extensive knowledge about multiple topics; however, there exist some specific needs for which this general knowledge is not enough. This is the case of specific tasks for a tutoring system constrained by a strict framework. For that reason, a specific technique called Fine-Tuning is used. Fine tuning is a machine learning technique in which a general-purpose LLM is trained over a smaller domain specific dataset to improve performance on a specific task [25].

A limitation of this process is the need for an extensive amount of data for the training process. To solve this, Knowledge Distillation is employed; it is a technique where a leading model with higher reasoning capabilities is used as a "Teacher" model, to transfer their capabilities into a smaller model called "Student" [26]; this is done for example by generating part of the input dataset from a powerful model.

In this project, fine-tuning was used to specialized an LLM for the learning framework specific task, and then knowledge distillation was applied to complete the initial dataset with more examples that can improve the fine-tuned model.

2.3.3 Knowledge Tracing and LSTM Networks

Knowledge tracing is a model in which student knowledge during a class is analyzed in order to predict future overcomes. In this model, historical interactions are modeled using deep learning techniques such as Recurrent Neural Networks (RNN) because they represent data over time [17]. Some of the recent techniques implement long- and Short-Term Memory (LSTM) networks to forecast the final results of the students. LSTM is a model part of RNN designed to learn over long sequences of data (in this case, students' interactions) but dealing with the traditional problem of short memory of RNN. LSTM models are based on a unique cell state with three gates (input, output, and forget) that select which information is retained or forgotten [27].

In this thesis, an LSTM model for knowledge tracing is used as part of the proposed tutoring system to evaluate students' interactions and predict a final outcome of the course.

3

Proposed Framework and System Architecture

This project proposes an architecture that integrates a cognitive diagnosis based on LLMs with knowledge tracing models through enriched pedagogical labels derived from a Neo-Piagetian learning framework. Its aim is to analyze how to diagnose the cognitive state of students in C programming courses using Large Language Models (LLMs) and the Neo-Piagetian framework, in order to generate tailored adaptive exercises that help students master C programming skills. This approach creates a learning flow in which, based on coding exercises, the LLMs are able to determine the most appropriate exercises for each student's cognitive stage, generating multiple student-model interactions. In addition, using Knowledge Tracing techniques, these interactions are translated to neural network inputs with the objective of predicting the risk of failure on the final exam. This entire learning workflow is finally translated to a web application in which students can interact with specialized LLMs in order to master specific C skills for their programming course.

This chapter details the two foundational pillars of the project: first, the integrated educational framework that maps programming topics to distinct cognitive stages; and second, the high-level system architecture that orchestrates the interactions between the *Diagnostic Engine*, the *Remediation Engine*, and the *Predictive Engine*.

3.1 Integrated Learning Framework

Based on the theoretical foundations established in the previous chapter, this section introduces the pedagogical core of the proposed system. Recognizing the complementary strengths of both approaches: Neo-Piaget Model and Block Model, this project proposes an integrated learning framework that combines the Neo-Piaget stages with the structural precision of the Block Model. By intersecting these frameworks, it is possible to map exactly what structural parts of the code a student is capable of understanding based on their current cognitive stage.

To this end, this project uses the analysis proposed by Izu et al. [22] in which the authors mention that specific programming activities designed to measure comprehension can be mapped into the dimensions of the Block Model. According to the authors, three foundational activities are indicators of a learner's understanding:

- **Tracing Task:** defined as following the execution of a program atom by atom. To successfully perform this task, the learner must first perceive the code as static text before understanding its execution. Consequently, mastering a tracing task indicates that the student has overcome the Text Surface (T) dimension of the Block Model.
- **Explain in your own words Task:** this task requires the student to explain in natural language the purpose of the code, which can be mapped as the Program Execution (P) and Function/Purpose (F) of the Block Model.
- **Parson Problems:** in this task, the correct code is provided to the student but it is broken into blocks that are not in the correct position, and the task is to arrange the blocks in the correct order to make the program run successfully. Successfully completing this activity requires understanding the macro-structure and relations within the code, representing the higher-level dimensions of the Block Model.

As can be seen, these three tasks can be easily mapped to the Neo-Piaget stages: sensorimotor, pre-operational, and concrete operational due to its shared emphasis on tracing, explaining, and writing code tasks. For that reason, table 3.1 shows a mapping between the Neo-Piaget stages and their corresponding horizontal dimension of the Block Model. By establishing this integrated framework mainly based on the cognitive stages of Piaget, we are able to map exactly which activities and specific tasks a student should be able to do in each stage.

Table 3.1: Mapping between Neo-Piagetian stages and the Block Model.

Cognitive Stage	Block Model Component
Sensorimotor (Pre-tracing)	Text Surface (T)
Pre-operational (Tracing)	Program Execution (P)
Concrete Operational (Post-tracing)	Function/Purpose (F)

3.2 Exercise Typologies Derived from the Integrated Learning Framework

Once the learning programming model was defined as a combination between Neo Piaget's cognitive stages [3] and the horizontal dimension of the Block Model [22], this framework will be used as part of the experiments with the LLMs in order to generate tailored exercises according to an progressive learning model for C programming. The use of the model is not just for the definition of the learning stages, but also to generate a list of adequate exercises according to each stage. For that reason, a list of exercise typologies was created for use, specifically in 5.4.1.

As mentioned in the last section, the foundation of this typology comes primarily from the study of the Block Model that represents how programming learning can be mapped in hierarchical steps of increasing complexity [22]. Each block corresponds to a group of cognitive skills, which leads to a progressive transition from basic syntax to higher-level problem-solving abstraction. To contextualize this Block structure within a cognitive development framework, the model was mapped to the three Neo-Piagetian stages: Sensorimotor, Pre-operational, and Concrete Operational, as defined in [3].

As part of this framework, some works ([28], [29]) were done to explore how learners acquire programming thinking skills according to their cognitive stage. Lister et al. [28] and López et al. [29] use Neo Piaget's stages to prepare academic examinations to analyze the performance of their students at the end of an introductory programming course. In these works, the authors show examples of different exercises used in each stage to validate whether a student has acquired a specific stage. For instance, Lister et al. [28] emphasize how a student in a pre-operational stage will give a line-by-line code explanation instead of an abstraction of the purpose of the code, as a concrete operational student would do. The authors use exercises like:

- Trace the value of a specific variable after the code has been executed for the pre-operational

stage.

- Explain the purpose of the code for the concrete operational stage.

On the other hand, López et al. [29] give a wider set of exercises for each stage. The work proposes exercises of the type:

- Match terms with definitions or find syntax errors for the sensorimotor stage.
- Missing lines in the code, where the student is asked to reorder the lines of the code to have a correct programming structure. This is part of the pre-operational stage.
- Parson problems, which refer to reorder lines of code, again in a pre-operational stage.
- Explain in plain English what a code does, which is part of the concrete operational stage.

Building on these theoretical foundations, a consolidated list of exercise types was developed (see Table 3.2). This categorization allows each type of exercise to be explicitly associated with a cognitive development phase, allowing them to be used as part of the LLMs prompt, with the objective of generating exercises that are aligned to the intended complexity of the learning framework.

Table 3.2: Exercise Types Mapped to Neo-Piagetian Cognitive Stages

Exercise Type	**Cognitive Stage**
Identify the parts of a syntax	Sensorimotor
Find syntax errors	Sensorimotor
Match terms to definitions	Sensorimotor
Parson Problems: reorder lines of code	Pre-operational
Identify the missing lines in a code	Pre-operational
Exception problems (scenarios with null values, memory leaks, etc.)	Pre-operational
Describe the purpose of the code	Concrete Operational
Explain in plain English	Concrete Operational
Identify equivalent blocks / Refactor	Concrete Operational
Determine redundant code	Concrete Operational

3.3 Topics and Skills

The final step in developing a foundational learning framework is to determine the specific domain knowledge required for the system to teach programming in C language. The existing literature provides various curriculum standards that were used as a basis to create a final list of topics the proposed system should teach. For instance, the work presented in [30] mentions the main knowledge areas for computer science education, where the main areas for programming are algorithms and programming languages. However, in order to obtain high precision in what the system should teach, the general programming topics must be broken down into atomic skills that can be measured by different exercises.

For that reason, the first task was to define the general topics, which were restricted to the fundamental concepts typically covered in the first half of a university-level C programming course (Conditionals, Loops, Arrays, etc.). Then, the atomic skills for each one of the topics were obtained with the help of an LLM like Claude Sonnet 4.5 and then manually reviewed together with a university professor teaching the C programming course in order to get a final list of topics and skills the system will use.

The next step was to map all these skills into a specific Neo-Piaget cognitive stage (sensorimotor, pre-operational, and concrete operational), with the aim of understanding the exercises the system must propose to overcome each skill based on the previous defined list (see table 3.2). With this process, the system will indicate to the students the current skill they are practicing and in which specific cognitive stage they are currently in. In the same way, this process was performed manually with the review of the expert university professor, table 3.3 shows the final list of topics and skills considered by the proposed system in this project.

Table 3.3: List of topics and skills, mapped to Neo-Piaget cognitive stages.

Cognitive Stage	Skill
	Expressions
Sensorimotor	S1: Recognize symbols and syntax of an expression.
Sensorimotor	S2: Distinguish between operators and operands.
Pre-operational	P1: Trace the value and type of arithmetic expressions.
Pre-operational	P2: Apply operator precedence and associativity when tracing.
Concrete Operational	C1: Combine multiple operators intentionally (e.g., (x + y) / 2.0).

Cognitive Stage	Skill
Concrete Operational	C2: Debug expression errors (e.g., wrong precedence, wrong type).
Concrete Operational	C3: Refactor expressions for clarity and efficiency.
	Logical Expressions
Sensorimotor	S1: Recognize logical operators in C (, ||, !).
Sensorimotor	S2: Distinguish logical operators from relational (>, <, ==) and arithmetic (+, -) operators.
Sensorimotor	S3: Recognize that operators have an order of evaluation (precedence).
Pre-operational	P1: Predict the Boolean result of logical expressions.
Pre-operational	P2: Combine relational and logical operators in tracing.
Pre-operational	P3: Trace expressions considering operator precedence.
Pre-operational	P4: Compare the result of expressions with and without parentheses.
Concrete Operational	C1: Combine multiple relational expressions with and || correctly.
Concrete Operational	C2: Refactor complex conditions into clearer logical expressions.
Concrete Operational	C3: Write expressions that rely on correct precedence without unnecessary parentheses.
Concrete Operational	C4: Debug logic errors caused by a misunderstanding of precedence.
	Variables
Sensorimotor	S1: Identify variable names (identifiers) and basic syntax of declarations.
Sensorimotor	S2: Recognize that different data types exist (int, float, char, etc.).
Sensorimotor	S3: Identify simple assignment statements (x = 5;).
Sensorimotor	S4: Understanding the lifespan of a variable
Pre-operational	P1: Trace how variable values change through assignment statements.
Pre-operational	P2: Understanding the "algorithm" executed by the C runtime for evaluating assignments (mostly related to the types of the result of the expression changes depending on the type of the variable)
Pre-operational	P3: Explain how data types affect value storage and type conversion.
Pre-operational	P4: Predict program output based on variable value changes and conversions.

Cognitive Stage	Skill
Concrete Operational	C1: Write correct assignment statements that update program state meaningfully.
Concrete Operational	C2: Choose suitable data types and apply explicit conversions when necessary.
Concrete Operational	C3: Manage variable scope and lifetime to avoid conflicts or memory issues.
	Conditionals
Sensorimotor	S1:Recognize the basic structure of an if statement.
Sensorimotor	S2:Identify keywords related to conditionals: if, else if, else.
Sensorimotor	S3:Identify the general syntax pattern of the ternary operator (condition ? expression1 : expression2).
Pre-operational	P1:Trace how a program decides which branch (if, else if, else or nested if) executes.
Pre-operational	P2:Trace how the ternary operator selects between two expressions.
Pre-operational	P3:Recognize common logical patterns used in conditions (>, <, ==, , ||).
Concrete Operational	C1:Determine what we know about the state of a program in each branch and after the execution of an IF statement. For example if(x>0) printf("yes") else printf("no"); while we are in the else branch we know that x<=0. After the if we know nothing about the value of x
Concrete Operational	C2:Write syntactically correct conditional statements and nested conditionals.
Concrete Operational	C3:Use the ternary operator to simplify simple conditional expressions.
Concrete Operational	C4:Design logical conditions that correctly represent a problem's requirements.
Concrete Operational	C5:Debug and refactor conditional statements for clarity and correctness.
	Loops
Sensorimotor	S1:Identify loop components: initialization, condition, update, and body.
Sensorimotor	S2:Distinguish loop keywords (while, for) from other C syntax elements.
Pre-operational	P1:Trace a loop step by step to follow variable changes and iteration flow.
Pre-operational	P2:Predict the number of iterations for a given simple loop.

Cognitive Stage	Skill
Pre-operational	P3:Compare the behavior of while and for loops in simple examples.
Pre-operational	P4:Identify errors in simple loop logic (e.g., infinite loop, off-by-one).
Concrete Operational	C1:What do we know about the state of the program before, during, and after the loop?
Concrete Operational	C2:Design and write loops that solve simple computational tasks.
Concrete Operational	C3:Choose the appropriate loop type (while, for) for different situations.
Concrete Operational	C4:Combine multiple loops effectively (nested loops) to process multi-level data.
Concrete Operational	C5:Optimize loop structure for clarity and efficiency.
	Functions
Sensorimotor	S1: Recognize the basic structure of a function in C (return_type name(parameters)).
Sensorimotor	S2: Distinguish between a function definition and a function call.
Sensorimotor	S3: Understand at a conceptual level that a function "does something" and can "give back a value."
Sensorimotor	S4: Recognize that variables have a scope (inside or outside functions).
Pre-operational	P1: Trace how control flows into and out of a function.
Pre-operational	P2: Identify and follow parameter passing.
Pre-operational	P3: Predict what value will be returned by a function based on the execution path.
Pre-operational	P4: Recognize and trace variable scope — which variables are visible inside or outside the function.
Pre-operational	P5: Observe the behavior of assert() in verifying conditions at runtime.
Concrete Operational	C1: Write and design functions that perform specific, reusable tasks.
Concrete Operational	C2: Select appropriate parameter types and return values for modular code design.
Concrete Operational	C3: Decompose complex problems into smaller, well-structured functions.
Concrete Operational	C4: Debug and reason about function behavior.
Concrete Operational	C5: Understand what a function computes
	Pointers

Cognitive Stage	Skill
Sensorimotor	S1: Recognize the syntax of pointer declarations.
Sensorimotor	S2: Distinguish between a variable and a pointer to a variable in written code.
Sensorimotor	S3: Understanding the meaning of * and & operators.
Sensorimotor	S4: Identify constant pointers and pointers to constants as different forms of declarations.
Sensorimotor	S5: Understanding pointer arithmetic.
Pre-operational	P1: Trace the value of a pointer and the value pointed to across program execution.
Pre-operational	P2: Predict how & obtains an address and how * dereferences it.
Pre-operational	P3: Follow changes in memory state when a pointer is assigned or dereferenced.
Pre-operational	P4: Trace the value of a pointer when it is passed to a function.
Pre-operational	P5: Trace situations involving operator precedence with pointers.
Pre-operational	P6: Identify and trace type compatibility issues.
Concrete Operational	C1: Write correct pointer declarations with appropriate types and qualifiers.
Concrete Operational	C2: Use &, *, and pointer arithmetic correctly, understanding operator precedence.
Concrete Operational	C3: Choose correct pointer types and ensure type compatibility in assignments and function parameters.
Concrete Operational	C4: Debug pointer-related issues such as invalid dereferencing, uninitialized pointers, and type mismatches.
	Arrays
Sensorimotor	S1: Recognize the syntax of array declarations.
Sensorimotor	S2: Identify that an array has a fixed size and cannot be resized.
Sensorimotor	S3: Distinguish array indexing visually.
Sensorimotor	S4: Recognize that strings in C are sequences of characters ending in '\0'.
Sensorimotor	S5: Notice that arrays and pointers share similar notation, without implying equivalence.
Sensorimotor	S6: Understand when arrays are cast to pointers (i.e. when used in expressions)

Cognitive Stage	Skill
Sensorimotor	S7: Notice that matrices use multiple brackets indicating "multiple dimensions".
Pre-operational	P1: Trace values inside an array using indices, including boundaries.
Pre-operational	P2: Trace what happens when an array is passed to a function.
Pre-operational	P3: Trace string operations in memory, including the null terminator.
Pre-operational	P4: Trace how character arrays differ from pointers to characters (char s[] vs char *s).
Pre-operational	P5: Trace the layout of multidimensional arrays row-by-row in memory.
Pre-operational	P6: Trace matrix access.
Pre-operational	P7: Identify out-of-bounds access when tracing.
Concrete Operational	C1: Declare and use arrays with correct boundaries and fixed sizes.
Concrete Operational	C2: Write functions that correctly receive arrays (via pointers) and manage length explicitly.
Concrete Operational	C3: Use character arrays and string pointers appropriately in typical operations (copying, iteration, input).
Concrete Operational	C4: Understand when arrays behave differently from pointers (sizeof, assignment, parameter passing).
Concrete Operational	C5: Work with multidimensional arrays and matrices using nested loops and/or pointer notation.
Concrete Operational	C6: Choose between array-based and pointer-based representations depending on the task.
Concrete Operational	C7: Avoid common errors: out-of-bounds access, missing '\0', incorrect function signatures.

3.4 System Architecture

The proposed autonomous system is designed to create a bridge between standard programming code submissions and personalized cognitive tutoring. Rather than being just a software interface, this system is an educational platform which combines three pillars: a data foundation, a core artificial intelligence pipeline, and a scalable web integration. Initially, a labeled data set is created in order to train and evaluate AI models. Then, the AI pipeline combines three

components: first, a *Diagnostic Engine* that uses specialized LLMs to analyze code submissions and determine the actual cognitive stage based on a Neo-Piagetian learning framework, based on predefined skills to each C programming topic. Second, a *Remediation Engine* that generates tailored exercises based on the previous cognitive diagnosis, with the aim of helping the student complete mastery of C programming topics. Third, a *Predictive Engine* which uses all the interactions between the student and the LLM to create a prediction model of the risk that the student fails the final exam. Finally, all the components are merged into a web application for future use.

Dataset Preparation: The first pillar of the architecture is the construction of a specialized dataset. This process begins with the extraction of raw educational logs from the Moodle platform of the introductory programming course of the Bachelor's Degree in Computer Science at the University of Padua. Specifically, logs from academic years 2020 to 2023 were utilized for model training, while the 2023-2024 period was isolated to ensure unbiased evaluation.

To transform these raw XML logs into useful information, the data presents a three-stage preparation process:

- **Manual Ground Truth Construction:** A subset of real student submissions was manually annotated with Neo-Piagetian cognitive stages and specific missing skills to establish a high-quality standard.
- **Knowledge Distillation:** To scale the labeling process to the thousands of remaining submissions, a knowledge distillation strategy was employed using a teacher model (Claude 4.5 Sonnet) to generate high-quality cognitive labels.
- **Synthetic Data Augmentation:** To address the inherent class imbalance in educational logs where correct submissions often exceed beginner errors, Sensorimotor and Pre-operational misconceptions were generated to complete the dataset by using a powerful LLM such as Claude Sonnet. After this process, the generated examples were manually reviewed to ensure consistency with the learning framework.

This resulting dataset provides the essential base that allows the system to recognize student patterns. With this cognitive foundation established, the architecture transitions from static data to the dynamic instructional workflow. The details of how the dataset was used are described in the next chapters.

The AI Pipeline: The central instructional workflow of the system presents an adaptive learning loop that triggers the moment a student logs into the web application. As presented in Figure 3.1, the student initially selects a specific C programming topic (see Section 3.3), and automatically the system employs an LLM to present an initial *global exercise*, which is a complete programming problem designed to evaluate all atomic skills of the selected topic. In that moment, our specialized LLM is triggered in order to analyze the code and evaluate the Neo-Piagetian stages, output the student current cognitive stage, and the skills the student has mastered or needs more practice. After that, using the diagnostic output of the last step, the system calls an LLM again in order to generate personalized exercises for each of the skills that the student needs to practice. This is the start of a learning loop in which the LLM generates multiple exercises until the student masters the current skill. These exercises presented are based on Neo-Piagetian techniques to evaluate the specific cognitive stage. Finally, the students will master all the skills of a specific C programming topic, which represents the moment to continue with the next topic in their curricula.

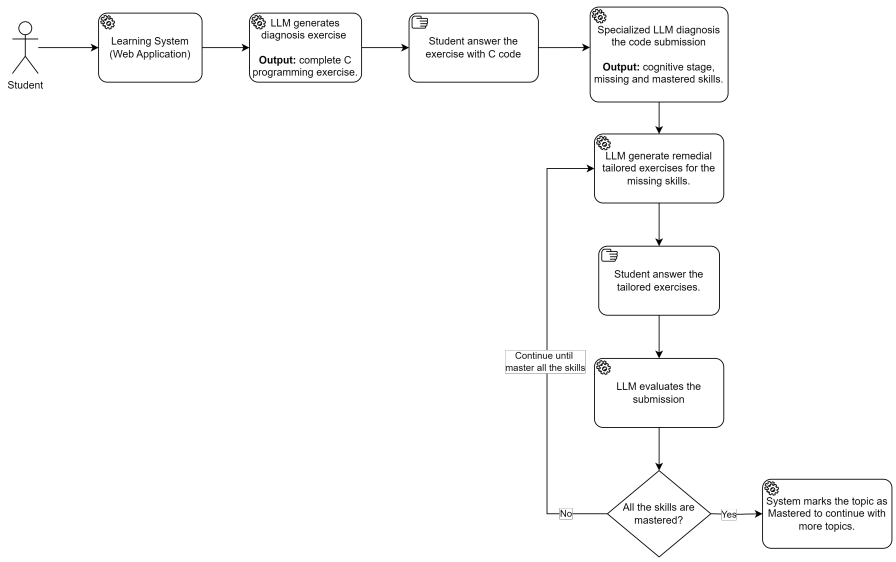

Figure 3.1: General System Workflow for Student interactions.

Beyond real-time exercise remediation and tutoring, the system is structured to continuously monitor student trajectories. Every interaction within the adaptive exercises loop is stored as a sequence of rich cognitive labels that contain the current Neo-Piaget stage, the mastered skills, and the missing skills, which are subsequently fed into the Knowledge Tracing model for its continuous risk prediction.

This intelligent workflow represents the brain of the system; however, for it to be accessible in a classroom environment, it requires a functional interface and a reliable infrastructure.

Web Application Integration: From an architectural perspective, the platform operates through a web application based on a client-server model that ensures real-time interaction, not only for students but also for professors and tutors. It uses React to build the front end and Python for the back end. As shown in Figure 3.2, the backend part uses FastAPI as a framework to orchestrate the workflow requests from the client side. In addition, the application uses Firebase Authentication and Firebase Database to store the users and their interactions in the tutoring system. In the same way, this orchestration allows the system to connect with the LLMs that are currently running in the University of Padua's cluster through the use of vLLM as a fast library for model inference [31].

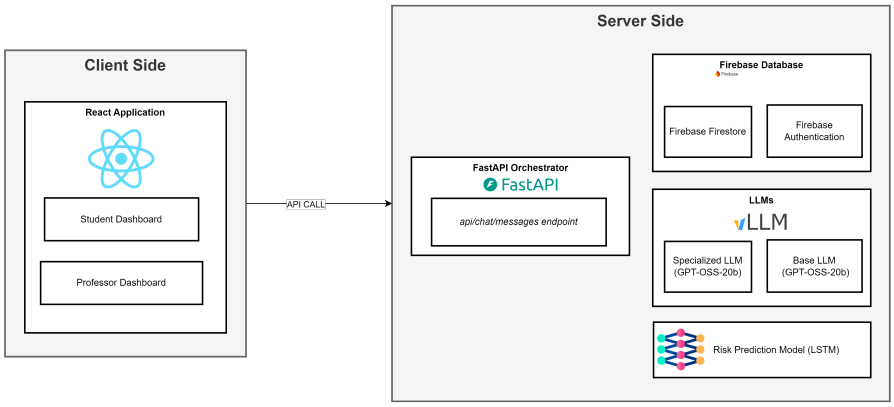

Figure 3.2: Proposed System Architecture

The final system presents two dashboards. The student dashboard presents all topics that are part of the programming course curriculum. In each topic, the system proposes an interactive chat connected to the LLMs to generate practice exercises as described in the workflow section; see Figure 3.3.

On the other hand, as shown in Figure 3.4, the professor dashboard presents the main statistics on students' interactions, as well as each student's predicted risk. In addition, the professor is able to see all the interactions made by the students.

Based on this architecture, the next sections present the technical implementation and the artificial intelligence mechanisms used to create the three engines of the main system.

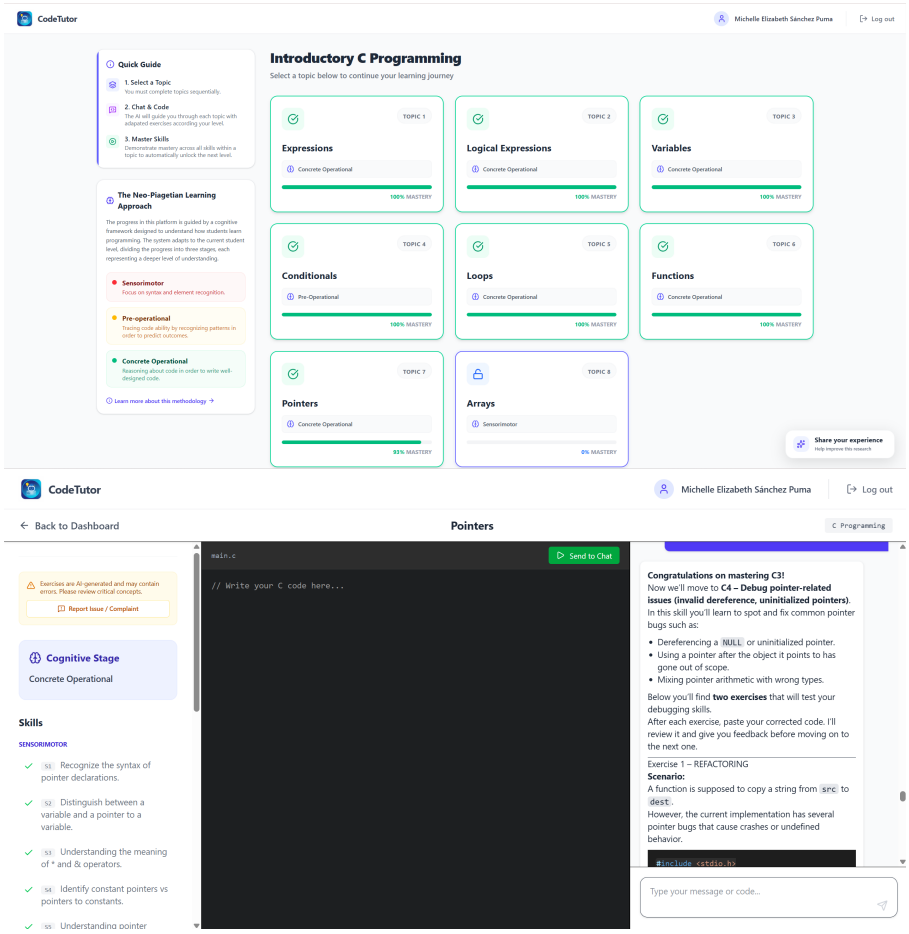

Figure 3.3: Student Dashboard of the proposed System.

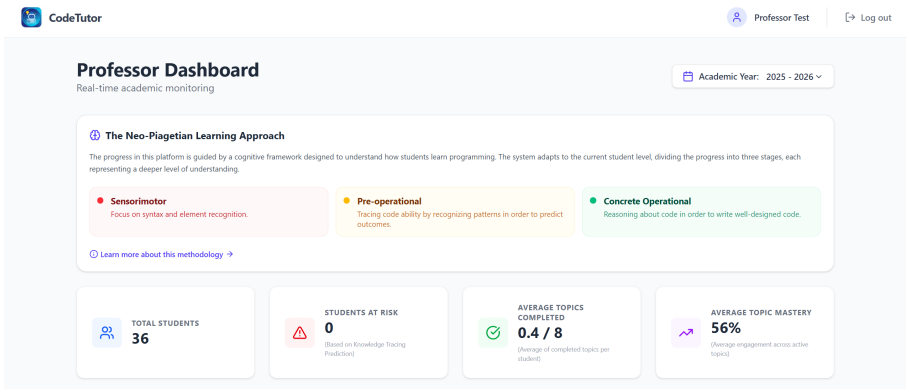

Figure 3.4: Professor Dashboard of the proposed System.

4

The Diagnostic Engine: Iterative Design and Evaluation

This chapter details the architecture and iterative development of the *Diagnostic Engine*, which is responsible for identifying students' missing skills based on their code submissions. Moving beyond standard exercise grading like pass or fail, this engine leverages the integrated Neo-Piaget learning framework discussed before to provide a complete diagnosis of a student's current understanding. This diagnosis is based on three main elements: the current cognitive stage, the skills the student has mastered, and the ones that the student needs to improve.

To achieve an accurate diagnostic tool, the engine began by using a prompt engineering process structured into iterative phases, starting from a foundational baseline prompt. Then, to obtain more advanced reasoning, a fine-tuning strategy was proposed. During the iterative phases, the architectures were analyzed using qualitative and quantitative metrics over different commercial and open-weight models like: GPT-5, Gemini 2.5-Flash, Claude Sonnet 4.5, GPT-OSS-20b, among others.

This chapter sequentially presents the methodological design and the corresponding experimental results for each development phase. This structure is designed to demonstrate how the quantitative limitations observed in early iterations motivated the architectural enhancements of the subsequent models.

4.1 Quantitative Evaluation Strategy

In order to evaluate the efficacy of Large Language Models in the diagnosis of missing and mastered skills, as well as the cognitive stage, this strategy aims to determine if those models could accurately diagnose why a student failed in a programming exercise, mapping their errors to specific Neo-Piagetian stages, and if the models could determine the final cognitive stage in which the student currently is. To achieve statistical validation, this evaluation follows a four-step pipeline.

- Construction of the ground-truth dataset.
- Manual labeling of the skills evaluated for each exercise in the dataset.
- Construction of the hybrid ground-truth dataset.
- Evaluate the efficacy of LLMs by comparing their responses to those of the ground-truth dataset.

Phase 1 - Construction of the ground-truth dataset: The first step of this large-scale evaluation is the construction of the ground-truth dataset, which was built using a hybrid approach. We began by analyzing the Moodle Logs of the Programming Course for the bachelor's degree in Computer Science at the University of Padua, which were provided by the faculty. These logs have information on homework, VPL (Virtual Programming Laboratories) assignments, and some tests taken from previous years (2020-2021, 2021-2022, 2022-2023, 2023-2024).

The logs contain data on real programming exercises in the C language and real students' responses, which provided authentic information with the potential to reflect actual learning curves. From this point on, a Python script was developed to read the XML files of the logs and extract information about some of the exercises and answers. In this way, multiple files with information about the exercise metadata (title, base code, among others) and multiple answers with the following information were obtained: student ID, answer code, and grade. Once we have the list of exercises, they were manually categorized according to the topic each was evaluating.

This was done to get exercises for each of the topics defined in Section 3.3, obtaining a total of 27 exercises and 229 students' answers as described in Table 4.1, the topics not included in the table were not present in the Moodle Logs exercises because the exercises were selected to evaluate a single topic.

Table 4.1: Exercises obtained by Moodle Log Analysis

Topic	# Exercises	# Answers
Expressions	0	0
Logical Expressions	0	0
Variables	2	17
Conditionals	3	23
Loops	9	72
Functions	2	18
Pointers	0	0
Arrays	11	99
Total	**27**	**229**

Phase 2 - Manual labeling of the skills evaluated: The second step was to manually label the skills each exercise from the Moodle logs was evaluating, and then define the skills each answer was mastering or missing; as there was not a massive dataset, the skills label were performed manually. The process was carried out using the topics and skills described in Section 3.3. At the end of this process, two datasets were created: one defines the metadata for the exercises, and the other defines the skills associated with each answer. The information of each one is presented in Table 4.2.

Table 4.2: Ground-truth Datasets for Large-Scale Evaluation

Exercises Metadata Dataset	EXERCISE_ID TOPIC SOURCE TITLE BASE_CODE EVALUATED_SKILLS
Students' Answers Dataset	EXERCISE_ID USER_ID GRADE ANSWER_CODE MASTERED_SKILLS MISSING_SKILLS COGNITIVE_STAGE

Phase 3 - Construction of the hybrid ground-truth dataset: The third step of this evaluation was to complete the current dataset with synthetic exercises generated by an LLM. This process aimed to ensure the dataset was complete enough to evaluate every possible cognitive stage and missing or mastered skill. This allowed us to simulate "bad student" personas by asking the LLM to introduce logic errors. To do so, the model Claude Sonnet 4.5 was used to generate 17 programming exercises and a total of 182 students' answers, divided into the topics explained before. As shown in Table 4.3, exercises for the Loops and Arrays topics were not generated because there was enough with the ones obtained from Moodle Logs.

Table 4.3: Exercises generated by Claude Sonnet 4.5

Topic	# Exercises	# Answers
Expressions	4	40
Logical Expressions	4	40
Variables	2	24
Conditionals	1	18
Loops	0	0
Functions	2	20
Pointers	4	40
Arrays	0	0
Total	**17**	**182**

Some research on using LLM-generated presents the use of simulated student profiles to generate responses to multiple-choice questions by defining the list of knowledge components that the student has mastered, has confusion about, or has no evidence of knowledge of [32]. This research paper found that the generative students proposed by the LLM produced logical responses that were aligned with their profiles. Using the same approach, multiple students' simulations were used as part of the ground-truth dataset [32].

The process of generating synthetic exercises and answers was done using two prompts. The first prompt (see figure 4.1) was used to ask the LLM to generate a certain number of exercises on a specific topic; in this case, the prompt includes the Neo-Piaget stages, as well as the set of skills for that topic. The prompt also asks the LLM to specify the set of skills the exercise is evaluating to be in agreement with the information required in the exercise's dataset. The second prompt (see figure 4.2) was aimed at generating customized responses for a specific exercise, as well as the mastered and missing skills. In the same way, this prompt uses the Neo-Piaget Stages information and the list of skills evaluated by each exercise.

Generate Exercises Prompt for Ground-Truth dataset

System Role: You are an expert programming tutor helping beginners understand C programming. Your task is to create {num_exercises} new non-trivial C programming exercises aligned with a specific topic, a set of skills, and a Piaget cognitive stage.

Input:
Topic: {topic}

Piaget Cognitive Model: The model follows three learning stages.
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how program flow works.
2. **Pre-operational** – Students trace and interpret the code's execution but struggle with abstraction.
3. **Concrete-operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Target Skills: You must use the following skills. {skills}

Task: Generate exercises that satisfy the following conditions.
- The exercise must be explicitly designed to evaluate all the **target skills** of the specific topic.
- Do not use other topics or skills outside the given list.
- Ensure the problem is non-trivial and requires understanding of the topic and skills.
- Evaluate all the skills in a balanced manner.
- Include a clear and short problem statement; do not be verbose and do not ask explanation questions.
- The exercise type must be a programming problem that requires writing C code.
- Write the complete, syntactically correct C code for the optimal solution.

Output (Strict JSON Format for each generated exercise):

```
 {
"exercise_id": "e.g. L1_01_POWER_FUNC",
"theme": "[Value from Input]",
"title": "[Value from Input]",
"problem_statement": "[The generated C programming problem statement]",
"base_code_template": "[Complete, correct C code with the main
```

```
function]",
"skills_evaluated_ground_truth": "[Comma-separated list of REQUIRED
skill IDs (e.g., S1,P1,C2,C4), NO SPACES]"
}
```

Figure 4.1: Generate Exercises Prompt for Ground-Truth dataset.

Generate Answers Prompt for Ground-Truth dataset

System Role: You are an automated Cognitive Error Simulator for a C programming educational application. Your task is to generate simulated student responses based on a specific exercise and a provided optimal solution. You must deliberately introduce programming flaws that reflect a lack of specific cognitive skills following a Piaget stages model.

Piaget Cognitive Model: The model follows three learning stages.
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how program flow works.
2. **Pre-operational** – Students trace and interpret the code's execution but struggle with abstraction.
3. **Concrete-operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Input:
Exercise: {exercise}
Topic: {topic}
Skills to Evaluate: {skills}
Optimal Solution: {solution}

Objective: Generate 10 distinct student responses for the given exercise, divided into two failure pools.
- Generate 4 answers where the student is missing a random subset of the **pre-operational** skills and therefore the **concrete-operational** skills. In this pool the **sensorimotor** skills are mastered.
- Generate 4 answers where the student is missing a random subset of the **concrete-operational** skills. In this pool the **sensorimotor** and **pre-operational** skills are mastered.
- Generate 2 answers where the student has mastered all skills.

Task: For each answer perform the following.
- Produce a realistic student response.
- Ensure the mistakes directly reflect the missing skills.
- Mark all remaining skills as mastered.
- Vary the errors across answers; do not repeat patterns.
- For each response, randomly select a few unique skills from the designated pool to mark as `missing_skills`.

Output Format (Strict JSON):

```
 [
{
"exercise_id": "{exercise_id}",
"student": "1234",
"grade": "[Grade for the answer between 0-100]",
"answer_code": "<answer text>",
"mastered_skills": "[Comma-separated list of mastered skill IDs (e.g.,
S1,P1,C2,C4), NO SPACES]",
"missing_skills": "[Comma-separated list of missing skill IDs (e.g.,
S1,P1,C2,C4), NO SPACES]"
},
{
"exercise_id": "{exercise_id}",
"student": "1234",
"grade": "[Grade for the answer between 0-100]",
"answer_code": "<answer text>",
"mastered_skills": "[Comma-separated list of mastered skill IDs (e.g.,
S1,P1,C2,C4), NO SPACES]",
"missing_skills": "[Comma-separated list of missing skill IDs (e.g.,
S1,P1,C2,C4), NO SPACES]"
}
]
```

Figure 4.2: Generate Answers Prompt for Ground-Truth dataset.

After this hybrid process to generate the ground-truth dataset, it was made up of 44 program-

ming exercises and 411 student submissions as described in Table 4.4. Finally, a CSV file was obtained with the exercises and answers for each. This was then converted to a JSON file organized by exercise ID and its corresponding answers.

Table 4.4: Final number of Exercises for the Ground-Truth dataset

Topic	# Exercises	# Answers
Expressions	4	40
Logical Expressions	4	40
Variables	4	41
Conditionals	4	41
Loops	9	72
Functions	4	38
Pointers	4	40
Arrays	11	99
Total	**44**	**411**

Phase 4 - Evaluate the efficacy of the LLMs: The fourth step of this process is to apply the evaluation strategy to all iterative methods of the *Diagnostic Engine*. To evaluate the efficacy of the model and its architecture, the results presented by the LLMs must be compared to the ground-truth dataset in order to determine how well the LLMs answered. This is done by comparing the 411 students' submissions diagnosis between those from the ground-truth and those from the LLM responses. Specifically, the process measures the accuracy of the models in determining the cognitive stage, missing, and mastered skills, creating an analysis of the results based in two areas: cognitive stage classification as general performance and skill granularity evaluation.

To formalize this comparative analysis, a set of classification metrics was employed, computed using the True Positives (*TP*), True Negatives (*TN*), False Positives (*FP*), and False Negatives (*FN*) derived from a Confusion Matrix:

- **Confusion Matrix:** Serves as the foundational tabular representation of the model's predictions versus the actual ground-truth labels. This metrics allows for a detailed visualization of the correct and incorrect classifications across the different cognitive stages of the Neo-Piaget framework.
- **Accuracy:** Measures the overall proportion of correct predictions out of the total sub-

missions evaluated, establishing the baseline performance used to compare the models.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision:** Evaluates the exactness of the model, specifically the proportion of true positive diagnoses among all instances the model predicted as positive. This metric is crucial because it ensures the model does not flag skills as mastered when they are not.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity):** Measures the completeness of the model, representing its ability to correctly identify all actual positive instances. In this context, it is essential to ensure that the model detects all the skills that are really missing.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:** Provides the harmonic mean of Precision and Recall. It offers a balanced evaluation metric that is useful for addressing the class imbalances present in the dataset.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

This evaluation pipeline is the base for measuring the performance of the iterative design architecture of the *Diagnostic Engine*, and it is useful to compare how well a model performs on the specific diagnosis task using the Neo-Piaget learning framework.

4.2 Baseline Prompt for Missing Skills Diagnostic

This section explores the first iteration of the diagnostic component. The following subsections detail the methodological design of the baseline prompt and its experimental results, which establish the initial performance evaluation of the system.

4.2.1 Methodology Design and Implementation

The initial phase of the *Diagnostic Engine* involved the development of a baseline prompt that tries to explore the ability of Large Language Models (LLMs) to perform cognitive diagnosis.

The primary objective of this baseline prompt was to evaluate how well a general-purpose LLM can identify the mastered and missing skills of a specific topic in C programming and their corresponding Neo-Piagetian stages, based only on a code submission.
As a preliminary qualitative experiment to assess how well the models perform overall if used "as is" for a complete tutoring session, a full interaction was tested. In this scenario, the model is asked to generate a complete and complex exercise on a specific C programming topic, wait for the student's answer, and then identify the errors the student made. Those errors are then mapped to each skill provided to the model, which also corresponds to a Neo-Piaget stage according to the teaching cognitive model. In that way, the model can identify the missing and mastered skills and the cognitive stage at which the student is currently at.
Based on this, the prompt presented in Figure 4.3 was designed with the following characteristics:

- The model employs a concise description of the Neo-Piaget cognitive model.
- The model utilizes a list of skills for a specific topic in C programming, where each atomic skill belongs to one Neo-Piaget stage, refer to section 3.3.
- The first task is to generate an exercise and determine the level of understanding of the student through its submitted solution, so that it can evaluate the submission of the code.
- When all skills are mastered, the LLM provides some recommendations for refactoring or coding best practices.

However, to rigorously evaluate the diagnostic task and isolate it from the exercise generation task, the specific quantitative evaluation is also described in the results section. This task was done by directly providing the LLM with predefined code solutions.

Baseline Prompt for Cognitive Framework Diagnostic

System Role: You are an expert programming tutor helping beginners understand {topic} in C programming. Your goal is to help students master the topic of {topic} in C by assessing their current skill level using Piaget's cognitive development stages.

Cognitive Model (Piaget): Students may be at one of the following stages:
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how program flow works.
2. **Pre-operational** – Students trace the code's execution but struggle with abstraction.

3. **Concrete-operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Skills for the Topic: Each skill is mapped to one of the Piaget's three stages. {skills}

Instructions:
- Create a complex and comprehensive exercise involving {topic} in C.
- Use the Piaget model to analyze the student's cognitive stage upon their response.
- Identify missing skills based on the student's answer.
- After analyzing a student's submission, validate your assessment in 1–2 lines.

Teaching Instructions:
- Begin with a friendly introduction, stating your goal and that you will assess their understanding of {topic} using Piaget's model.
- Present the exercise to the student and wait for the student's answer.
- Analyze the answer and assess the student's cognitive stage and any missing skills.
- Report the identified mastered skill(s), the missing skill(s), current cognitive stage, and a concise explanation of your reasoning.
- If all the skills are mastered, offer the student recommendations about refactoring for efficiency and best practices.

Figure 4.3: Baseline Prompt for Cognitive Framework Diagnostic.

4.2.2 Baseline Results

To evaluate the efficacy of the previously described architecture, the baseline prompt was tested using five models: ChatGPT (GPT-5), Gemini 2.5 Pro, Claude 4.5 Sonnet, Llama 4, and GPT-OSS-20B. The evaluation process begins with a qualitative analysis of the reasoning capacity of each LLM, which was divided into two distinct scenarios.

1. **Error-Injected Submissions:** Models were presented with flawed code answers to measure their ability to correctly identify missing skills and diagnose a lower cognitive stage.
2. **Error-Free Submissions (Perfect exercise):** The models were presented with optimal code answers to assess their precision, specifically testing if they could recognize fully mastered skills without hallucinating false positive errors.

The qualitative observations for each model under these conditions are detailed below:

ChatGPT (GPT-5): GPT demonstrates a solid starting point by explaining the cognitive model. However, the generated exercise is relatively simple compared to those produced by other models, which indicates that this exercise is not good enough to measure all skills. In terms of the result, its reasoning regarding missing skills is coherent and consistent; the overall depth of explanation is not as detailed as in other systems. In the "perfect exercise" scenario, GPT correctly recognizes that the student has mastered all the target skills but does not propose specific examples of code refactoring or good programming practices that could extend the learning. [1]

Gemini 2.5-Flash: Gemini provides an excellent introduction, clearly explaining the role of the model and the learning framework. The initial exercise is well-constructed and cognitively appropriate, with a large complexity, trying to use all the skills. In the results, the identification of missing skills is both accurate and reasoned in a pedagogical way, with in-depth analysis of each skill. In the "perfect exercise" case, Gemini stands out by not only confirming the mastery of all skills but also providing thoughtful recommendations and best practices that reinforce the quality of the code. [2]

Claude (Sonnet 4.5): Claude begins with a strong introduction, explaining well the teaching model and the corresponding skills. Its proposed exercise is more comprehensive and integrates all the skills. Additionally, the model accurately places the learner within the Neo-Piagetian framework and offers a detailed explanation of missing skills. Claude clearly examines every skill in the "perfect exercise" scenario and makes specific suggestions that are pertinent to the assignment. In general, Claude exhibits the most complete behavior among all tested models. [3]

Llama 4 (llama-4-scout-17b-16e-instruct): Although Llama 4 has a strong start, it does not provide a detailed explanation of the learning model's cognitive phases. Its exercises remain very basic, showing limited alignment with the cognitive progression of the learning framework. On the positive side, the model provides an adequate analysis of mastered and missing skills,

[1] Complete interaction for the GPT-5 experiment can be accessed at `https://drive.google.com/drive/folders/1GZTDBvXiBLuoQlfA_5mAeJLtWnB06NIb?usp=sharing`.

[2] Complete interaction for the Gemini2.5-Flash experiment can be accessed at `https://drive.google.com/drive/folders/1n3wn6NY5iq18-W2-R69w-gbxnyGxh12j?usp=sharing`.

[3] Complete interaction for the Claude Sonnet 4.5 experiment can be accessed at `https://drive.google.com/drive/folders/1dOrKOcxLaZAV6uCKTeQDh5NZ3IKZ_0tm?usp=sharing`.

along with constructive recommendations at the end-of the task. Nevertheless, it incorrectly concludes that the student still lacks certain skills from the concrete operational stage in the "perfect exercise" scenario. [4]

GPT-OSS (gpt-oss-120b): GPT-OSS starts with a clear and accurate explanation of the teaching model and presents an exercise of higher complexity than other models. Its analysis of missing skills is precise and supports learning by providing the correct final version of the code requested. However, the learner's cognitive stage is explained in less detail, and it could use some additional clarification. GPT-OSS excels in the "perfect exercise" scenario, specifically acknowledging skill mastery and adding contextually relevant examples of code refactoring and best practices to its feedback. [5]

In conclusion, the qualitative analysis shows that Gemini 2.5-Flash and Claude Sonnet 4.5 both give the best performances and both give useful feedback. GPT-OSS also works very well, especially when it comes to taking technical feedback into account. GPT-5 does not have much analytical depth or flexibility. On the other hand, Llama 4 struggles in understanding the cognitive model and complexity of the task. The strengths and weaknesses of the models are summarized in Table 4.5.
Although qualitative observations provided valuable information on the reasoning capabilities of each LLM, this analysis is not sufficient to validate their Neo-Piaget diagnostic reliability. To complement these findings, a quantitative analysis was conducted following the evaluation methodology defined in Section 4.1. This evaluation process is based on, using the same exercises, comparing the responses of the LLM to the ground-truth diagnosis; thus, the models were tested using metrics such as accuracy, precision, recall and F1-Score to analyze the overall accuracy of the models to detect missing and mastered skills, as well as their classification accuracy of the Neo-Piaget cognitive stages. This evaluation process was performed for each one of the tested models as presented subsequently.

Gemini 2.5 - Flash: This model demonstrated a moderate ability to classify students into the correct Neo-Piaget stage. It achieves an accuracy of 65.21% and a weighted F1-score of 63.47% between all topics when detecting the final cognitive stage. The highest reliability was observed

[4]Complete interaction for the Llama 4 experiment can be accessed at `https://drive.google.com/drive/folders/1jARQrWGJoOZ301ev4cb--rXu0m9WGF_G?usp=sharing`.

[5]Complete interaction for the GPT-OSS-120b experiment can be accessed at `https://drive.google.com/drive/folders/1jTY2-VsbBEEHuUn7FNhIT4eonXJaly9U?usp=sharing`.

Table 4.5: Strengths and Weaknesses of the Models in Identifying Missing Skills Test

Model	Strengths	Weaknesses
GPT-5	Demonstrates coherent reasoning about missing skills.	The proposed exercise is overly simple and lacks depth compared to other models.
Gemini 2.5 Flash	Exercises are complex, identifies missing skills accurately and provides rich feedback.	-
Claude Sonnet 4.5	Provides the better output. Integrates all skills coherently and provides detailed, context-relevant recommendations.	-
Llama 4 Scout 17B	-	Exercises are too basic and it incorrectly identifies missing skills in the "perfect exercise" scenario.
GPT-OSS 120B	Generates exercises of high complexity and precise analysis of missing skills.	-

in the Pointers topic with an accuracy of 75%. This suggests that the model is particularly effective at isolating misconceptions and structural errors related to more complex memory management concepts. The values are presented in Table 4.6 and Figure 4.4.

Table 4.6: Quantitative Evaluation for Gemini 2.5 Flash Model - Metrics by Cognitive Stage

Topic	Accuracy	F1-Score
Total All Topics	0.6521	0.6347
Expressions	0.5750	0.5333
Logical Expressions	0.6750	0.6192
Variables	0.6341	0.5919
Conditionals	0.6341	0.6740
Loops	0.6806	0.6550
Functions	0.4474	0.4301
Pointers	0.7500	0.6860
Arrays	0.7071	0.6975

By analyzing the confusion matrix (see Figure 4.5) it is demonstrated that while the model generally avoids diagnosing higher stages for low-level errors, it tends to make more classifications

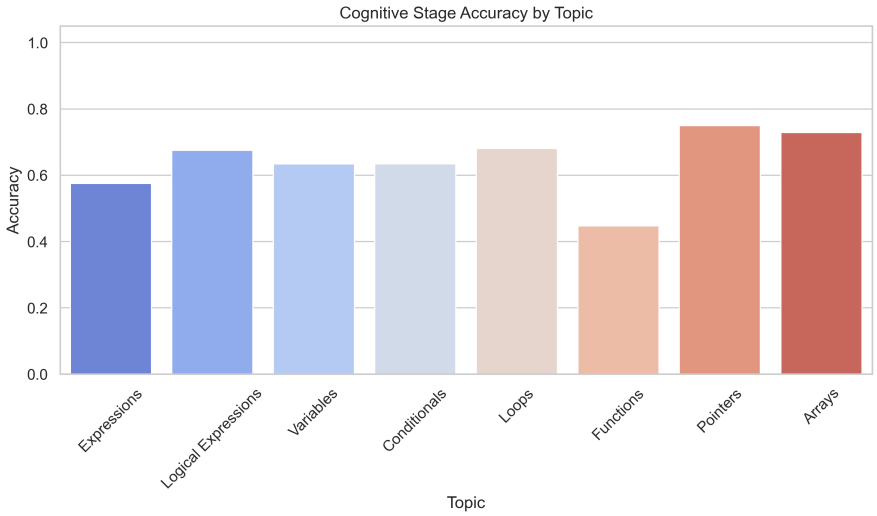

Figure 4.4: Accuracy for Baseline Prompt in Gemini 2.5-Flash Model

into the Pre-operational stage, which means it is underestimating the number of students at the Concrete Operational level.

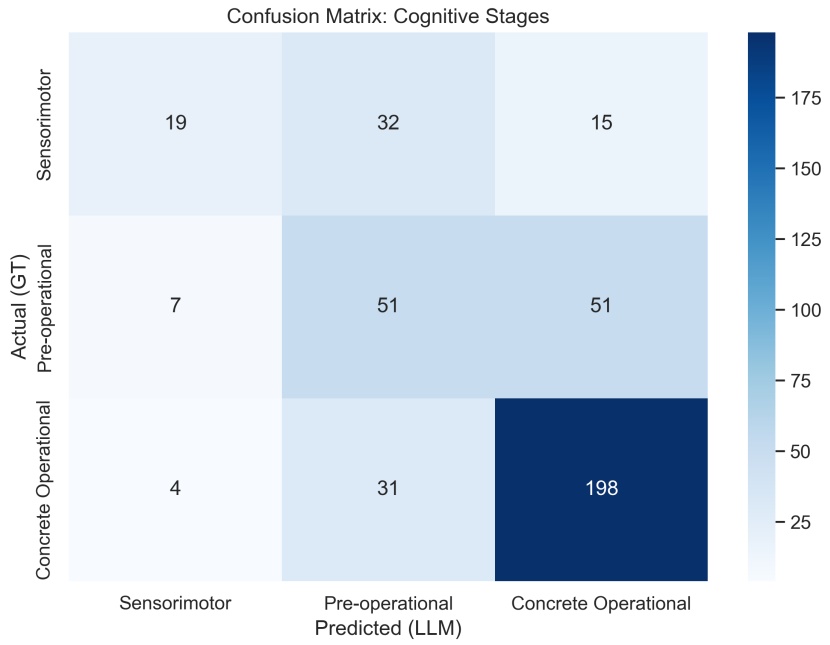

Figure 4.5: Confusion Matrix for Baseline prompt in Gemini 2.5 Flash Model

In addition, a multi-label analysis of individual skills was performed in order to identify the

concrete strengths and weaknesses of the model. Table 4.7 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

Table 4.7: Quantitative Evaluation for Gemini 2.5 Flash Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.807	0.861	0.585	0.697
MA-Expressions	0.836	0.840	0.908	0.873
MI-Logical Expressions	0.748	0.690	0.301	0.419
MA-Logical Expressions	0.777	0.785	0.938	0.855
MI-Variables	0.803	0.673	0.343	0.454
MA-Variables	0.805	0.811	0.940	0.871
MI-Conditionals	0.840	0.758	0.595	0.667
MA-Conditionals	0.834	0.833	0.837	0.835
MI-Loops	0.874	0.628	0.623	0.626
MA-Loops	0.874	0.905	0.899	0.902
MI-Functions	0.729	0.544	0.439	0.486
MA-Functions	0.718	0.743	0.799	0.770
MI-Pointers	0.783	0.771	0.430	0.552
MA-Pointers	0.790	0.804	0.911	0.854
MI-Arrays	0.859	0.527	0.552	0.539
MA-Arrays	0.846	0.841	0.760	0.799

First, we analyzed the performance of identifying skills as missing skills, as shown in Figure 4.6, and the model has a precision of almost 100% in all skills of the sensorimotor stage. However, depending on the topic and the specific skill, the model presents some problems. For instance, skill C2 of the Logical Expressions Topic has an accuracy of 57.5%, which indicates that the model misses half of all real errors. This is also supported by the low recall percentage of the missing skills in this topic, which indicates that the LLM is not doing a good job of recognizing the missing skills, i.e., the errors made by the students. In contrast, skills from the Pre-operational stage (P #) have an accuracy of more than 70%, which means that the model can correctly evaluate whether a tracking skill is missing in an answer.

Analysis of skill mastery revealed that this validation is generally greater than missing diagnoses. For example, almost all skills of all topics have an accuracy greater than 60%, except for the skill C5 of the function topic, which means that it is generally difficult to detect if it is mastered or even present in the exercise (see Figure 4.7).

In conclusion, this model demonstrates significant potential as a programming tutor, achieving a 65.21% of reliability in classifying the overall learning stage, with the Pre-operational stage as

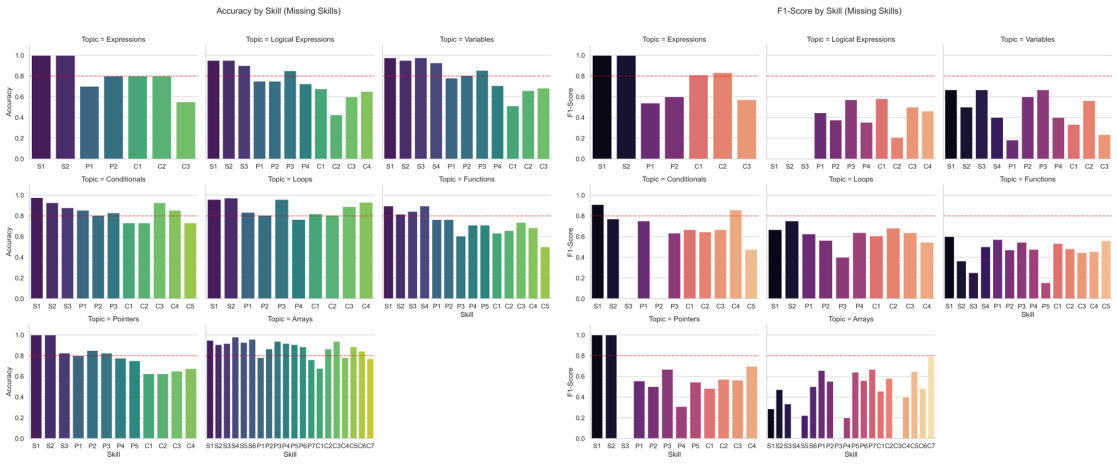

Figure 4.6: Accuracy and F1-score by Missing Skill for Baseline Prompt in Gemini 2.5 Flash Model.

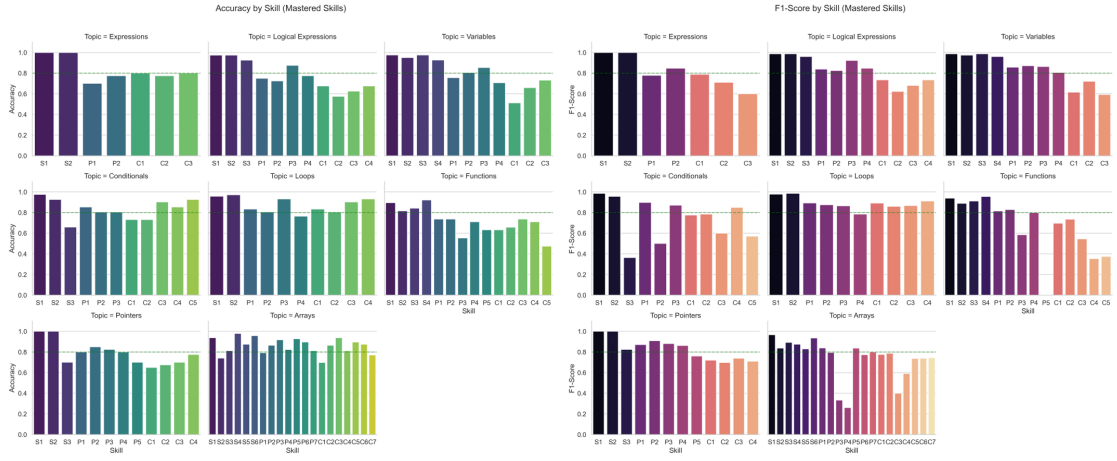

Figure 4.7: Accuracy and F1-score by Mastered Skill for Baseline Prompt in Gemini 2.5 Flash Model.

the best validation (see Figure 4.8). In general, the model's greatest strength is the basic pattern recognition of Sensorimotor Skills, but its major flaw is in determining correctly when a student is effectively in an advanced skill, especially when this Concrete Operational skill is related to debugging or refactoring tasks.

GPT-OSS (gpt-oss-120b): This model demonstrated a moderate ability to classify students into the correct Neo-Piaget stage. It achieves an accuracy of 63.26% and a weighted F1-score of 61.30% between all topics when detecting the final cognitive stage. The highest reliability was observed in the conditionals topic with an accuracy of 70.73%, suggesting that this model is better in this topic compared to the Gemini 2.5-Flash Model. The values are presented in

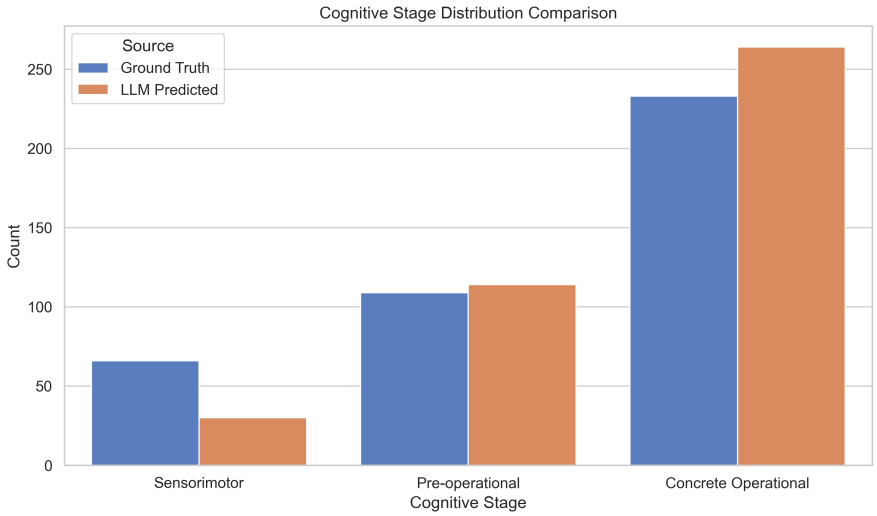

Figure 4.8: Stage Distribution for Baseline Prompt in Gemini 2.5 - Flash Model.

Table 4.8 and Figure 4.9.

Table 4.8: Quantitative Evaluation for GPT-OSS-120b Model - Metrics by Cognitive Stage

Topic	Accuracy	F1-Score
Total All Topics	0.6326	0.6130
Expressions	0.5750	0.5810
Logical Expressions	0.6500	0.6039
Variables	0.6098	0.5589
Conditionals	0.7073	0.7236
Loops	0.6667	0.6169
Functions	0.6316	0.5617
Pointers	0.6500	0.6319
Arrays	0.5960	0.6111

By analyzing the confusion matrix (see Figure 4.10), it is demonstrated that the model tends to do a good job diagnosing the higher stages (Concrete Operational Stage). Compared with the Gemini 2.5-Flash model, it does not tend to estimate a higher number of students at the Pre-operational level.

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model. Table 4.9 shows the metrics for each topic, considering the analysis of identified missing and

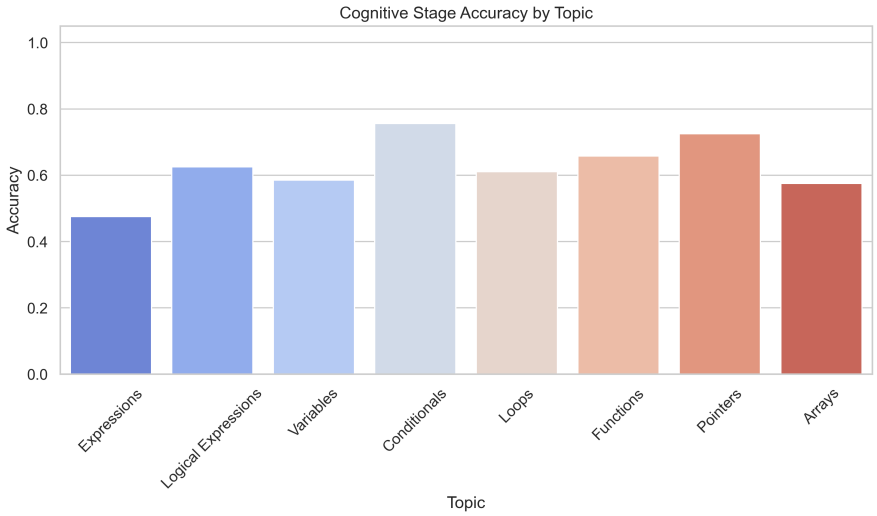

Figure 4.9: Accuracy for Baseline Prompt in GPT-OSS-120b Model

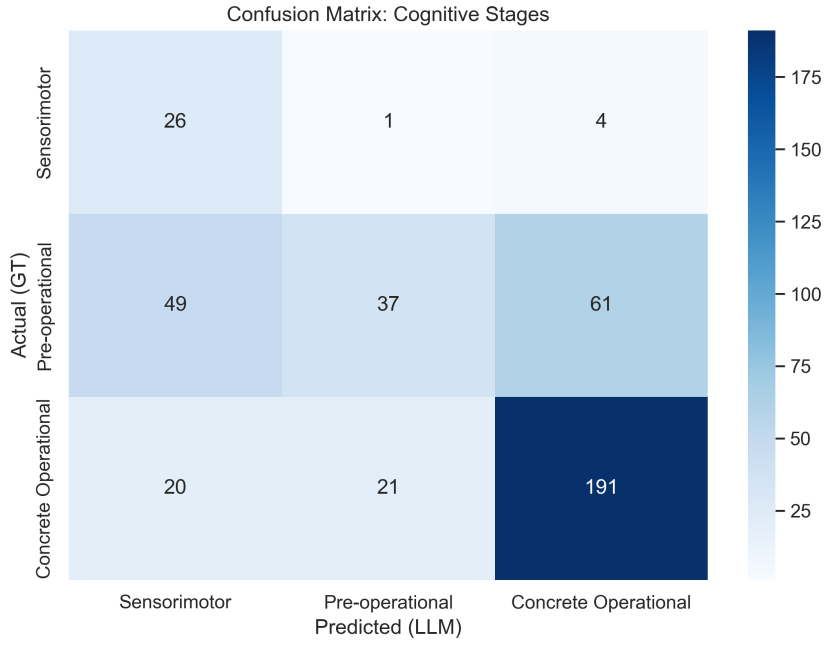

Figure 4.10: Confusion Matrix for Baseline Prompt in GPT-OSS-120b Model

mastered skills.

First, we analyzed the performance of identifying skills as missing skills, as shown in Figure 4.11. The table shows the metrics for each topic, considering the analysis of the identified

Table 4.9: Quantitative Evaluation for GPT-OSS-120b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.768	0.647	0.849	0.735
MA-Expressions	0.775	0.899	0.718	0.799
MI-Logical Expressions	0.668	0.469	0.737	0.573
MA-Logical Expressions	0.668	0.861	0.625	0.725
MI-Variables	0.716	0.436	0.630	0.515
MA-Variables	0.721	0.859	0.717	0.782
MI-Conditionals	0.825	0.652	0.744	0.695
MA-Conditionals	0.812	0.878	0.727	0.795
MI-Loops	0.724	0.366	0.861	0.513
MA-Loops	0.724	0.944	0.610	0.741
MI-Functions	0.795	0.838	0.368	0.511
MA-Functions	0.784	0.743	0.968	0.841
MI-Pointers	0.723	0.547	0.631	0.586
MA-Pointers	0.698	0.819	0.711	0.761
MI-Arrays	0.797	0.413	0.842	0.554
MA-Arrays	0.796	0.897	0.557	0.687

missing and mastered skills. This model also has an accuracy of almost 100% in all skills of the Sensorimotor Stage. However, depending on the topic and the specific skill, the model presents some problems. For instance, skills P1 and P2 of the Loops Topic have an accuracy of 55.6% and 43.1%, respectively, which indicates that the model is not good at determining if those skills are missing in a specific tracing exercise. This is also seen in the low precision of the missing skills in this topic with a value of 36.6%. The same scenario we have with the skill C5 of the Functions topic, which has an accuracy of 44.74%, demonstrates that the model does not recognize well that the skill is missing in a programming code answer. In this model, the Preoperational and Concrete operational stages are more stable in comparison with the Gemini 2.5-Flash model, as all of them are not overestimated.

Analysis of skill mastery revealed that this validation is not as good as the one done by the Gemini model because, in this case, not all the skills have a higher accuracy. In this case, almost all skills have an accuracy greater than 60%, but others have a lower accuracy, even close to 40%, which means that it is generally difficult to detect if it is mastered or even present in the exercise (see Figure 4.12).

In conclusion, this model demonstrates a similar potential as a programming tutor compared to Gemini due to its consistency in the Concrete Operational stage, achieving a reliability of

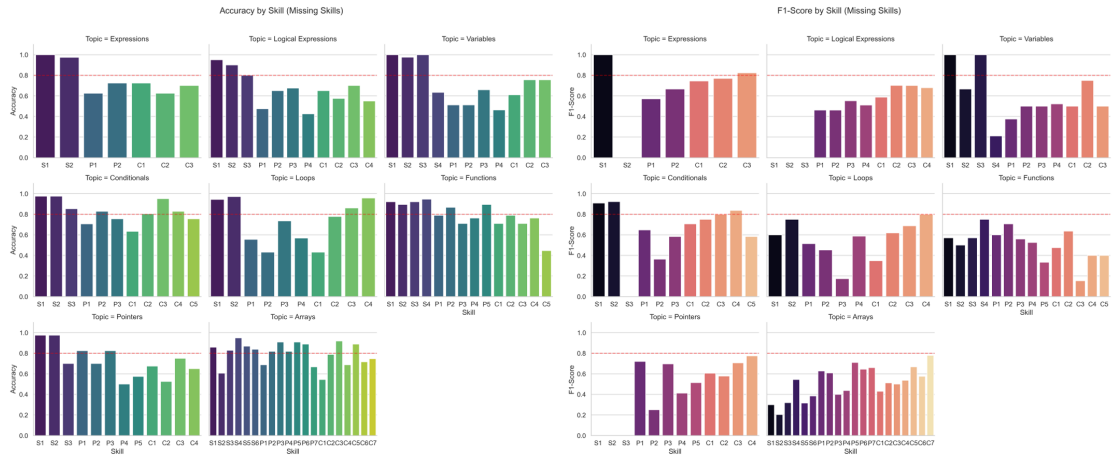

Figure 4.11: Accuracy by Missing Skill for Baseline Prompt in GPT-OSS-120b Model

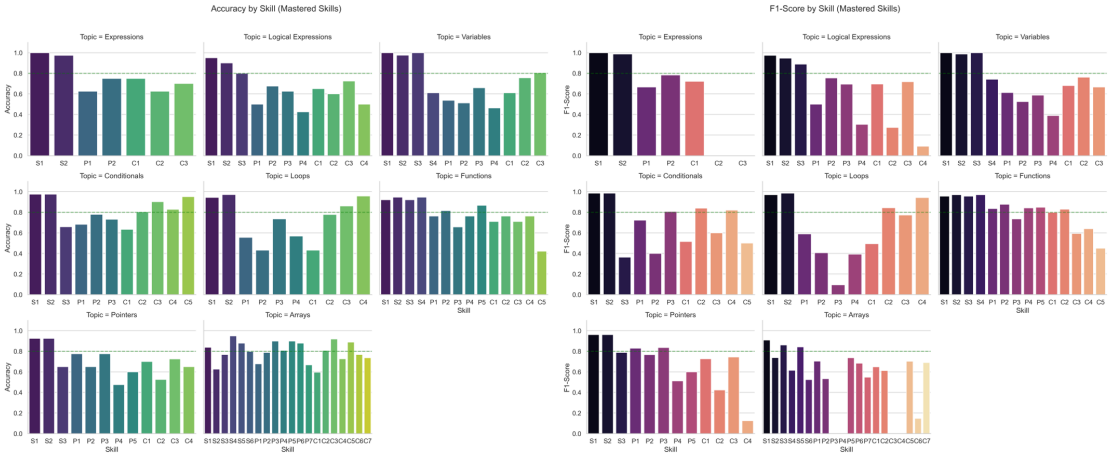

Figure 4.12: Accuracy by Mastered Skill for Baseline Prompt in GPT-OSS-120b Model

63.26% in classifying the overall learning stage, with the Concrete Operational stage as the best validation (see Figure 4.13). In general, the model's greatest strength is the basic pattern recognition of Sensorimotor Skills, but its major flaw is in determining correctly when a student is between a Pre-operational or a Concrete-operational stage.

GPT-OSS (gpt-oss-20b): This model again demonstrated a moderate ability to classify students into the correct Neo-Piaget stage. It achieves an accuracy of 64.48% and a weighted F1-score of 62.49% between all topics when detecting the final cognitive stage. The highest reliability was observed in the Pointers topic with an accuracy of 72.50%, suggesting that this model is better at recognizing this specific ability. The values are presented in Table 4.10 and

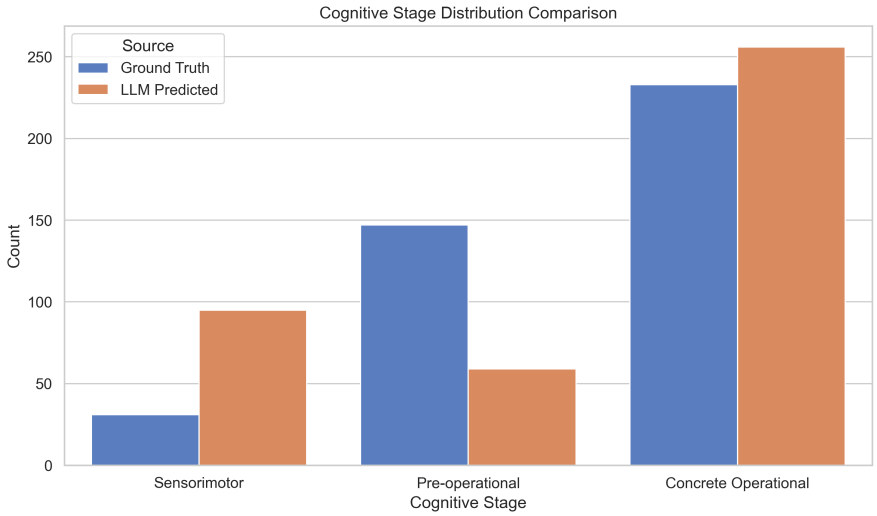

Figure 4.13: Stage Distribution of the Large-Scale Evaluation for GPT-oSS-120b Model.

Figure 4.14.

Table 4.10: Quantitative Evaluation for GPT-OSS-20b Model - Metrics by Cognitive Stage

Topic	Accuracy	F1-Score
Total All Topics	0.6448	0.6249
Expressions	0.5750	0.5810
Logical Expressions	0.6500	0.6039
Variables	0.6098	0.5589
Conditionals	0.7073	0.7236
Loops	0.6667	0.6169
Functions	0.6316	0.5617
Pointers	0.7250	0.7049
Arrays	0.6162	0.6319

By analyzing the confusion matrix (see Figure 4.15), it is demonstrated that the model tends to do a good job diagnosing the higher stages (Concrete Operational Stage). Compared with the Gemini 2.5-Flash model, it does not tend to estimate a higher number of students at the Pre-operational level. This is almost the same evaluation as GPT-OSS-120b did, which indicates that the models are very similar in terms of this evaluation.
The multi-label analysis of individual skills identifies the strengths and weaknesses of the model.

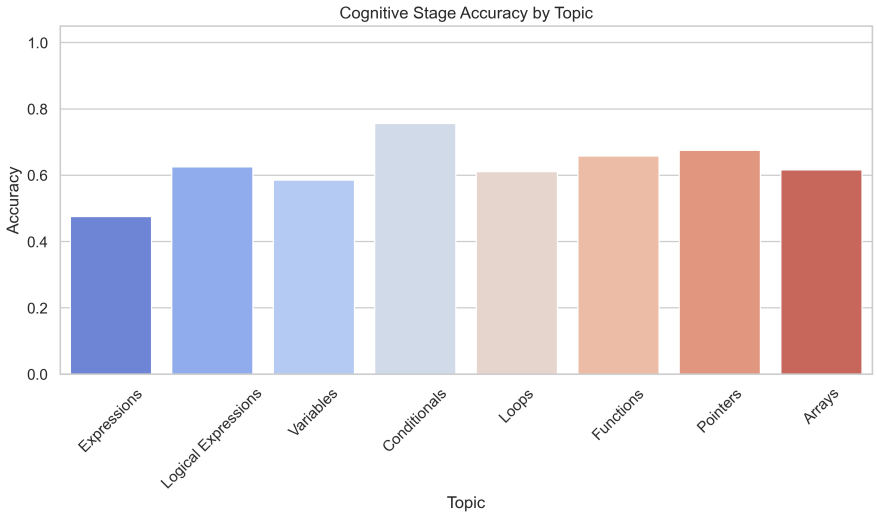

Figure 4.14: Accuracy for Baseline Prompt in GPT-OSS-20b Model

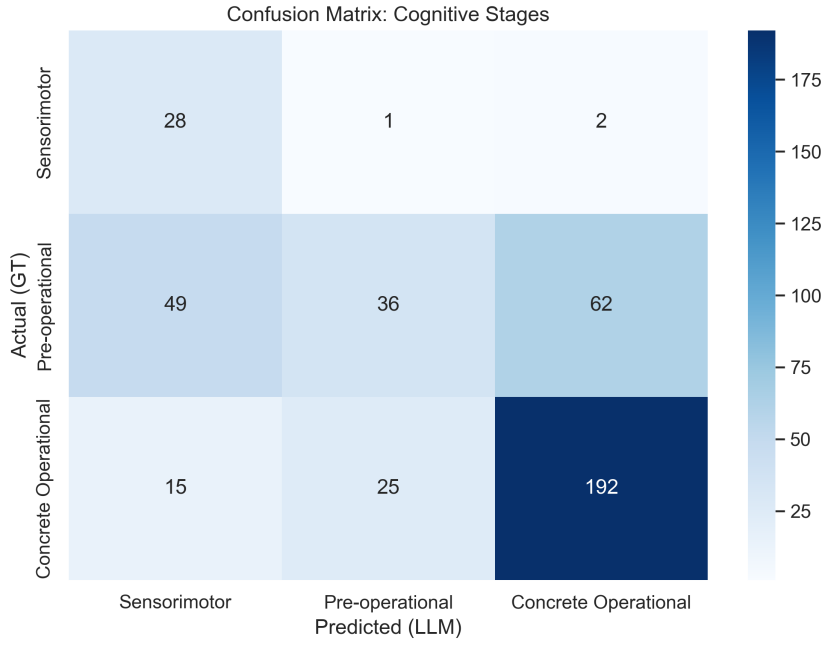

Figure 4.15: Confusion Matrix for Large-Scale Evaluation in GPT-OSS-20b Model

Table 4.11 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

First, as in the other models, we analyzed the performance of identifying skills as missing skills,

Table 4.11: Quantitative Evaluation for GPT-OSS-20b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.768	0.647	0.849	0.735
MA-Expressions	0.775	0.899	0.718	0.799
MI-Logical Expressions	0.668	0.469	0.737	0.573
MA-Logical Expressions	0.668	0.861	0.625	0.725
MI-Variables	0.716	0.436	0.630	0.515
MA-Variables	0.721	0.859	0.717	0.782
MI-Conditionals	0.825	0.652	0.744	0.695
MA-Conditionals	0.812	0.878	0.727	0.795
MI-Loops	0.724	0.366	0.861	0.513
MA-Loops	0.724	0.944	0.610	0.741
MI-Functions	0.795	0.838	0.368	0.511
MA-Functions	0.784	0.743	0.968	0.841
MI-Pointers	0.704	0.526	0.470	0.496
MA-Pointers	0.690	0.777	0.760	0.768
MI-Arrays	0.811	0.434	0.865	0.578
MA-Arrays	0.801	0.943	0.536	0.684

as shown in Figure 4.16. This model also has an accuracy of almost 100% in all skills of the Sensorimotor Stage. However, depending on the topic and the specific skill, the model presents some problems. For instance, skill P4 of the Logical Expressions Topic has an accuracy of 42.50%, which indicates that the model is not good at determining if that skill is missing in a tracing exercise. The same scenario we have with the skill C5 of the Functions topic, which has an accuracy of 42.1%, complemented with a low recall of 36.8% on that topic. In this model, the Pre-operational and Concrete operational stages are more stable in comparison to the other two models, as most of the Concrete Operational skills have a higher accuracy.

Analysis of mastered skills revealed that, similar to the GPT-OSS-120B model, this validation is not as good as the one performed by the Gemini model because. However, it is better than the GPT-OSS-120b one because almost all skills have an accuracy greater than 60%; the exceptions are skills related to Logical Expressions and Functions, which means that they are generally difficult to detect if those skills are mastered or even present in the exercise (see Figure 4.17).

In conclusion, this model demonstrates a very similar potential as a programming tutor compared to GPT-OSS-120b, which means they have basically the same reasoning abilities. This model achieves a reliability of 64.48% in classifying the overall learning stage, with the Concrete Operational stage as the best validation (see Figure 4.18). In general, the model's greatest

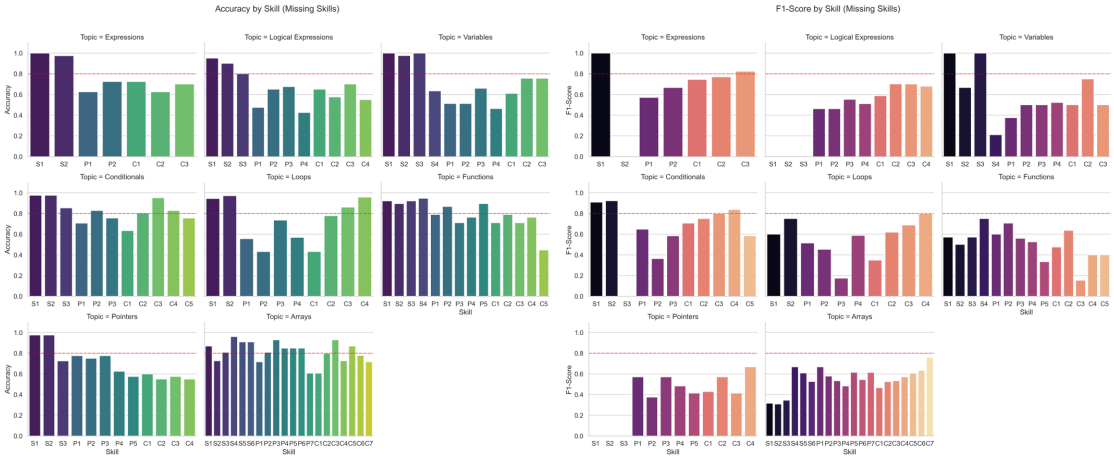

Figure 4.16: Accuracy by Missing Skill for Baseline Prompt in GPT-OSS-20b Model

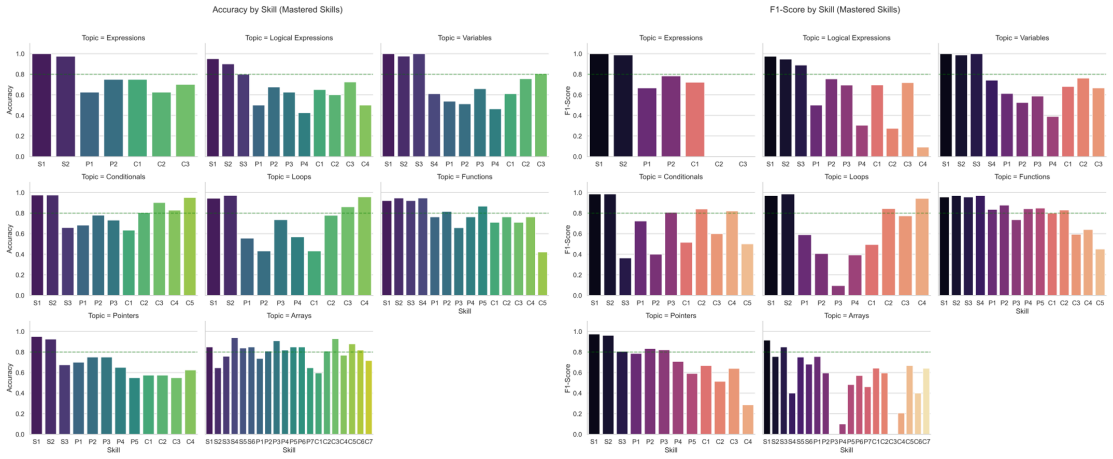

Figure 4.17: Accuracy by Mastered Skill for Baseline Prompt in GPT-OSS-20b Model

strength is the basic pattern recognition of Sensorimotor Skills and the good performance in recognizing a student in the Concrete Operational stage, but its major flaw is in determining correctly when a student is just in the Pre-operational stage.

To finalize the experiments of the baseline prompt, Table 4.12 summarizes the accuracy and weighted F1-score of the three models. All of them have an accuracy of nearly 65%, which indicates that the baseline architecture described in section 4.2.1 has the potential to be a diagnostic tool for programming skills; showing its strengths and limitations. Across all models, we could identify a consistent diagnostic pattern, where the Precision measure was generally high, indicating that the LLMs are excellent for identifying basic errors and highly reliable when they

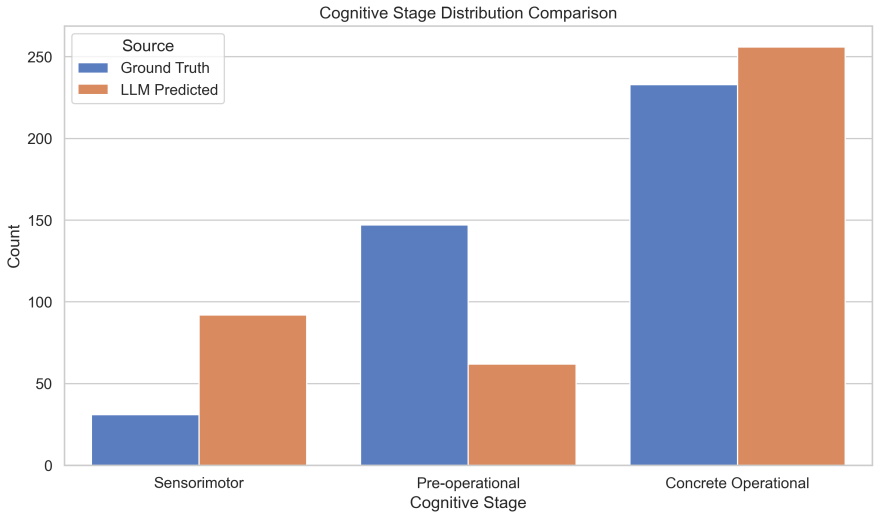

Figure 4.18: Stage Distribution for Baseline Prompt in GPT-oSS-20b Model.

indicate a student has a missing skill in their answer code. However, all models exhibited a persistent tendency to have problems in identifying more complex skills in the Concrete Operational stage, resulting in lower Recall for complex mistakes. In conclusion, this suggests that while the LLM is effective in diagnosing basic and intermediate atomic skills, more abstract concepts such as debugging or refactoring skills are still a limitation of models.

Table 4.12: Accuracy and F1-score for all models in the Large-Scale Evaluation Prompt

	Gemini 2.5 Flash		GPT-OSS 120b		GPT-OSS 20b	
	Acc	F1	Acc	F1	Acc	F1
Total All Topics	0.65	0.63	0.63	0.61	0.64	0.62
Expressions	0.58	0.53	0.58	0.58	0.58	0.58
Logical Expr.	0.67	0.62	0.65	0.60	0.65	0.60
Variables	0.63	0.59	0.61	0.56	0.61	0.56
Conditionals	0.63	0.67	0.71	0.72	0.71	0.72
Loops	0.68	0.65	0.67	0.62	0.67	0.62
Functions	0.45	0.43	0.63	0.56	0.63	0.56
Pointers	0.75	0.69	0.65	0.63	0.73	0.70
Arrays	0.71	0.70	0.60	0.61	0.62	0.63

This approach demonstrated that LLMs have the potential to diagnose basic stages, but present limitations on how models can effectively identify the correct stage of cognitive development.

In particular, the model struggled to reliably deduce the deeper cognitive stages of the learning framework. For that reason, it was necessary to create a more sophisticated prompt capable of guiding the LLM through a structured reasoning process.

4.3 Improving Baseline Prompt through Enhanced Prompt Engineering

This section explores the second iteration of the diagnostic component. The following subsections detail the methodological design of the enhanced prompt and its experimental results.

4.3.1 Methodology Design and Implementation

To address the reasoning shortcomings of the baseline prompt, the second iteration of the *Diagnostic Engine* incorporated advanced prompt engineering techniques. This improvement is to have the most robust and reliable analysis of cognitive diagnosis.
Instead of simply instructing the model to output a final classification, the enhanced prompt was redesigned to force the LLM to generate its analytical process step-by-step. The new prompt shown in the figure4.19 introduces the following key architectural features:

- Tips on how to identify each cognitive stage according to the learning framework.
- **Chain of thought:** The prompt asks the model first to recognize the error in the code answer and then assess the error to the correct skill, with the objective of having a step-by-step analysis.
- **Error Detection:** Ask the model to identify the type of error, i.e. if the answer has a syntax error probably the student is at Sensorimotor Stage; if the code has a simple logical error, the stage would be Pre-operational Stage; and if the student was able to create a correct code but it has refactoring or optimization issues, the student is probably in the Concrete Operational Stage.

Chain of Thought Strategy Prompt for Cognitive Framework Diagnostic

System Role: You are an expert C programming tutor for the topic {topic} who acts as an Automated Diagnostic System. Your goal is to assess a student's C programming submission based on Piaget's cognitive stages and a specific set of skills.

Title: {title}
Topic: {topic}

Cognitive Model (Piaget Stages): Analyze the code bottom-up. The **lowest level of error determines the stage**.
1. **Sensorimotor:**
Focus: Students recognize syntax and structure but do not yet reason about how program flow works.
Indicators: Syntax errors, typos, misunderstanding of basic keywords, not tracing abilities.
2. **Pre-operational:**
Focus: Students trace and interpret the code's execution but struggle with abstraction.
Indicators: Code compiles but logic is flawed due to poor understanding of control flow, variable updates, or tracing errors.
3. **Concrete-operational:**
Focus: Students understand the abstract logic and can apply the topic in problem-solving.
Indicators: The student uses the correct structures and logic to solve the problem; they may still have refactoring or optimization issues.

Skills: Each skill maps to a cognitive stage. Use the IDs provided (e.g., S1).
{skills}

Instructions:
- The exercise description and context have already been provided in the conversation history.
- Use the skills list to evaluate each submission.
- Use the Piaget model to analyze the student's cognitive stage upon their response.
- You must follow this algorithm strictly:
1. **Detect Errors:** Find the lowest level error (Syntax → Logic → Solution).
2. **Identify Missing:** Select the specific Skill IDs from the list above that caused the error.
3. **Identify Mastered:** All other skills from the provided list that are not missing are automatically **MASTERED**.
- You must categorize every single skill from the list as either Missing or Mastered. No skill can be left out of the final output JSON.
4. **Determine Stage:** Based on the lowest level error, determine the cognitive stage using the skills list and its corresponding cognitive stage.
5. **Format:** Output a raw JSON object first, then your text response.

Teaching Instructions:
- Begin with a friendly introduction.
- Report the identified mastered skill(s), the missing skill(s), current cognitive stage, and a concise explanation of your reasoning based on the JSON above.
- If all the skills are mastered, offer the student recommendations about refactoring for efficiency and best practices.

Output Format: First output the JSON object and then continue with the teaching instructions.

```
 {
"cognitive_stage": "Sensorimotor" | "Pre-operational" | "Concrete
Operational",
"missing_skills": ["S1", "P2"] or ["None"],
"mastered_skills": ["S2", "C1"] or ["None"]
}
```

Figure 4.19: Chain of Thought Strategy Prompt for Cognitive Framework Diagnostic.

4.3.2 Enhanced Prompt Results

In order to improve the results obtained for the baseline prompt, this new experiment was again tested in the models previously used, i.e., Gemini 2.5-Flash, GPT-OSS-120b and GPT-OSS-20b using the same methodology for large-scale evaluation. As described in Section 4.1, the evaluation is performed using a ground-truth dataset previously defined using the real exercises and students' responses from the provided Moodle logs. This ground-truth dataset was manually labeled with the learning framework information: missing skills, mastered skills, and cognitive stage. Consequently, the work of the LLMs is to provide the same labels (missing skills, mastered skills, and cognitive stage) for the exercises provided using the present prompt techniques. Finally, a comparison was made between ground-truth and LLMs evaluation to obtain a statistical analysis of performance.

Gemini 2.5-Flash: This model presents better performance compared to the test in section 4.2.2. Using this new prompt, the model achieves a total accuracy of 71.78% and a weighted F1 score of 71%, which represents an improvement considering the accuracy of 65.21% obtained

in the last test. Analyzing each topic, the evaluation of the "Pointers" topic revealed the best accuracy with 87.5%, these results are shown in Table 4.13.

Table 4.13: Quantitative Evaluation for Gemini 2.5-Flash Model using the enhanced prompt - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.7178	0.7100
Expressions	0.8000	0.8000
Logical Expressions	0.7250	0.7016
Variables	0.6829	0.6558
Conditionals	0.7073	0.7218
Loops	0.7361	0.7147
Functions	0.4474	0.4149
Pointers	0.8750	0.8704
Arrays	0.7273	0.7201

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model. Table 4.14 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

Table 4.14: Quantitative Evaluation for Gemini 2.5-Flash Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.807	0.861	0.585	0.697
MA-Expressions	0.836	0.840	0.908	0.873
MI-Logical Expressions	0.748	0.690	0.301	0.419
MA-Logical Expressions	0.777	0.785	0.938	0.855
MI-Variables	0.803	0.673	0.343	0.454
MA-Variables	0.805	0.811	0.940	0.871
MI-Conditionals	0.840	0.758	0.595	0.667
MA-Conditionals	0.834	0.833	0.837	0.835
MI-Loops	0.874	0.628	0.623	0.626
MA-Loops	0.874	0.905	0.899	0.902
MI-Functions	0.729	0.544	0.439	0.486
MA-Functions	0.718	0.743	0.799	0.770
MI-Pointers	0.783	0.771	0.430	0.552
MA-Pointers	0.790	0.804	0.911	0.854
MI-Arrays	0.859	0.527	0.552	0.539
MA-Arrays	0.846	0.841	0.760	0.799

This analysis shows that in general the precision of the model in all the topics is higher than in the last test. However, there are some recall values that could be improved; for instance, the identification of missing skills in the Logical Expressions topic, which has a recall of 30.1%. As shown in Figure 4.20, the confusion matrix of the results shows that the model is doing a good job recognizing the Pre-operational and Concrete Operational stages, not assigning a large number of students to these categories if they do not belong to them. In conclusion, the improvements made in the prompt allow the Gemini 2.5-Flash model to achieve a better identification of the missing and mastered skills, obtaining a considerable increase in its accuracy, compared to the baseline prompt used in the last test.

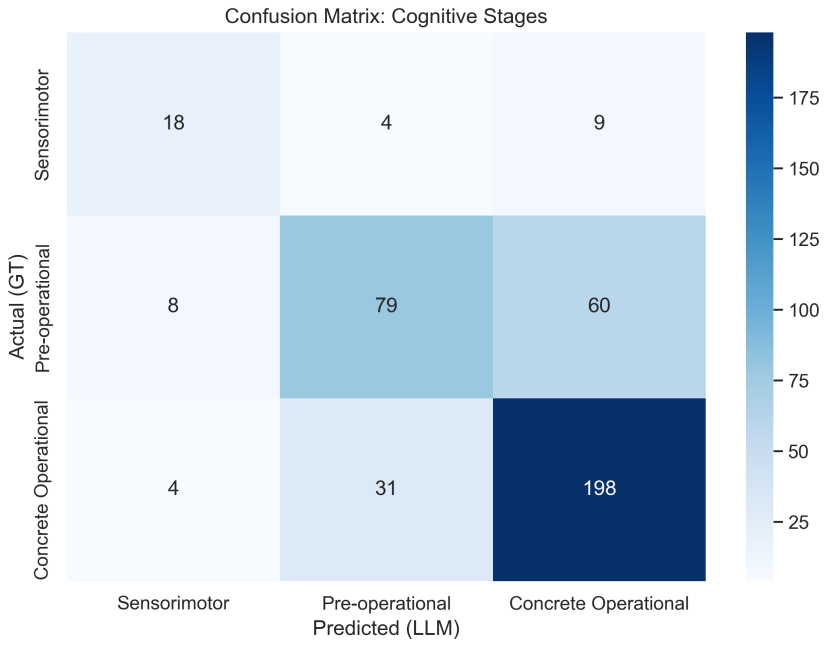

Figure 4.20: Confusion Matrix for Enhanced Prompt in Gemini 2.5-Flash model.

GPT-OSS-120b: As with the Gemini model, this model also presents better performance compared to the test in section 4.2.2. Using this enhanced prompt, the model achieves a total accuracy of 68.61% and a weighted F1 score of 70.63%, which represents an improvement considering the accuracy of 63.26% obtained in the last test. Analyzing each topic, the evaluation of the "Expressions" topic revealed the best accuracy with 90%, these results are shown in Table 4.15.

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model.

Table 4.15: Quantitative Evaluation for GPT-OSS-120b Model using the enhanced prompt - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.6861	0.7063
Expressions	0.9000	0.9003
Logical Expressions	0.8000	0.8088
Variables	0.6098	0.6243
Conditionals	0.6098	0.6375
Loops	0.7222	0.7293
Functions	0.7632	0.7627
Pointers	0.7000	0.7719
Arrays	0.5556	0.5982

Table 4.16 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

Table 4.16: Quantitative Evaluation for GPT-OSS-120b Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.800	0.917	0.519	0.663
MA-Expressions	0.800	0.768	0.971	0.858
MI-Logical Expressions	0.768	0.667	0.466	0.549
MA-Logical Expressions	0.768	0.795	0.899	0.844
MI-Variables	0.747	0.474	0.509	0.491
MA-Variables	0.749	0.832	0.803	0.817
MI-Conditionals	0.818	0.705	0.554	0.620
MA-Conditionals	0.798	0.793	0.811	0.802
MI-Loops	0.800	0.430	0.557	0.486
MA-Loops	0.800	0.875	0.807	0.840
MI-Functions	0.803	0.784	0.445	0.568
MA-Functions	0.806	0.771	0.955	0.853
MI-Pointers	0.704	0.559	0.221	0.317
MA-Pointers	0.708	0.720	0.932	0.812
MI-Arrays	0.799	0.392	0.613	0.478
MA-Arrays	0.803	0.805	0.673	0.733

This analysis shows that in general the precision of the model in all the topics is higher than in the last test. However, there are a few recall values that could be improved; for instance, the identification of missing skills in the Pointers topic, which has a recall of 22.1%. As shown in

Figure 4.21, the confusion matrix of the results shows that the model is doing a good job recognizing the Pre-operational and Concrete Operational stages, even better than Gemini 2.5-Flash due to is good recognizing of Pre-operational stages. In conclusion, the improvements made in the prompt allow the model to achieve a better identification of the missing and mastered skills, obtaining a considerable increase in its accuracy, compared to the baseline prompt used in the last test; although these values could be improved, it has an overall good performance.

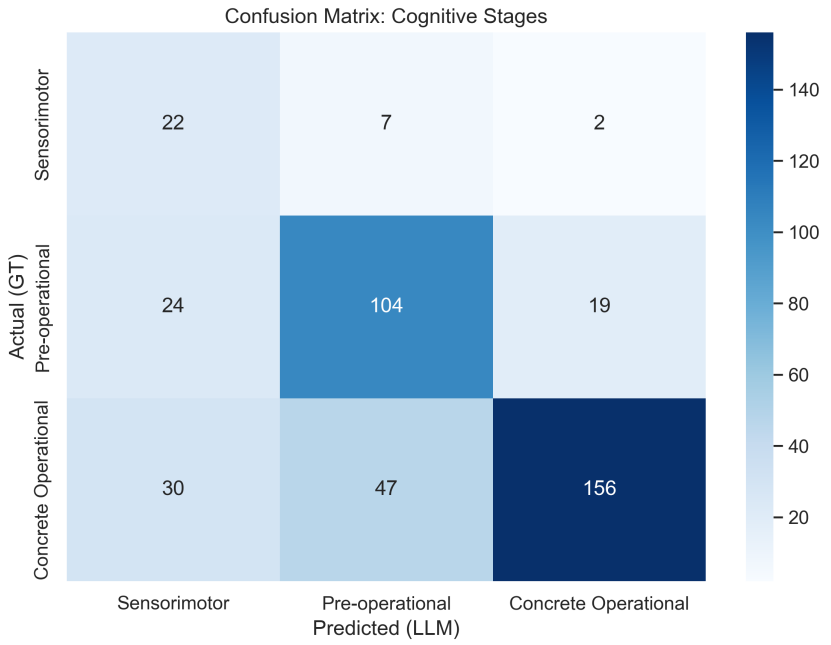

Figure 4.21: Confusion Matrix for Enhanced Prompt in GPT-OSS-120b model.

GPT-OSS-20b: Although this model presents better performance compared to the test in section 4.2.2, it does not have a significant increase in accuracy as the other models. Using this enhanced prompt, the model achieves a total accuracy of 65.94% and a weighted F1 score of 67.56%, which represents an improvement considering the accuracy of 64.48% obtained in the last test. Analyzing each topic, the evaluation of the "Expressions" topic revealed the best accuracy with 85%, these results are shown in Table 4.17.

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model. Table 4.18 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

This analysis shows that almost all the precision values are higher than in the last test, except

Table 4.17: Quantitative Evaluation for GPT-OSS-20b Model using the enhanced prompt - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.6594	0.6756
Expressions	0.8500	0.8500
Logical Expressions	0.8000	0.8040
Variables	0.5610	0.5802
Conditionals	0.6341	0.6604
Loops	0.6806	0.6878
Functions	0.7105	0.7088
Pointers	0.5750	0.6167
Arrays	0.5758	0.5978

Table 4.18: Quantitative Evaluation for GPT-OSS-20b Model using the enhanced prompt - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.707	0.761	0.330	0.461
MA-Expressions	0.707	0.697	0.937	0.799
MI-Logical Expressions	0.745	0.627	0.391	0.481
MA-Logical Expressions	0.745	0.773	0.899	0.831
MI-Variables	0.758	0.496	0.528	0.511
MA-Variables	0.761	0.839	0.813	0.826
MI-Conditionals	0.818	0.689	0.587	0.634
MA-Conditionals	0.800	0.807	0.793	0.800
MI-Loops	0.853	0.565	0.574	0.569
MA-Loops	0.847	0.887	0.876	0.881
MI-Functions	0.805	0.807	0.432	0.563
MA-Functions	0.816	0.784	0.949	0.859
MI-Pointers	0.719	0.617	0.248	0.354
MA-Pointers	0.715	0.724	0.935	0.816
MI-Arrays	0.784	0.352	0.525	0.422
MA-Arrays	0.778	0.761	0.652	0.702

for the identification of missing skills of the topics Variables and Arrays. In addition, there are some recall values that could be improved; for instance, the identification of missing skills in the Expressions topic, which has a recall of 33%. As shown in Figure 4.22, the confusion matrix of the results shows that the model does a good job recognizing the Pre-operational and Concrete Operational stages, but it is not as good as the GPT-OSS-120b model. In conclusion, the improvements made in the prompt allow the model to achieve a better identification of the

missing and mastered skills. Its results are not as good as the Gemini 2.5-Flash model due to the reduced reasoning capacity this open-weight model has; however, in this test, the prompt engineering strategy increases its performance.

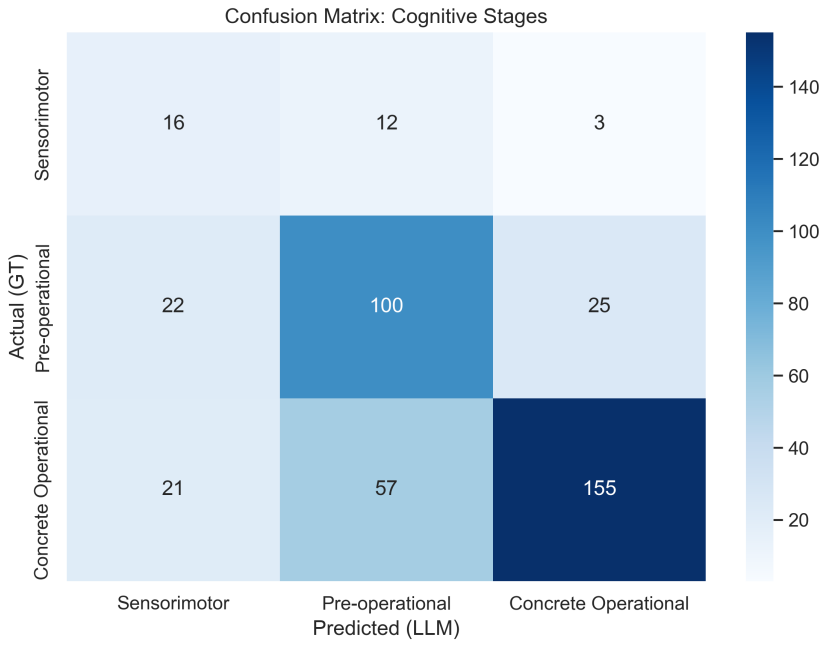

Figure 4.22: Confusion Matrix for Enhanced in GPT-OSS-20b model.

Finally, in this experiment, we analyzed how the overall performance of the models could increase using chain of thought techniques. The complete comparison could be reviewed in Table 4.19. In general, Gemini 2.5-Flash has considerably higher accuracy, showing good reasoning according to the learning framework. Furthermore, the GPT-OSS-120b model, thanks to its higher number of parameters, increases its accuracy and results in a better job of recognizing the evaluated cognitive stages. Although GPT-OSS-20b has also increased its accuracy, it did not achieve the same level of evaluation compared to the other two models; this is due to the limited number of parameters and its open-weight model condition.

In conclusion, although the Chain-of-Thought (CoT) strategy significantly improved the recognition of basic syntax errors, qualitative and quantitative observations indicated persistent challenges in differentiating complex errors at the highest cognitive stages. Consequently, relying solely on prompt engineering techniques is insufficient. For that reason, a more sophisticated architecture must be design in order for the model to learn specific information about the learning framework.

Table 4.19: Accuracy and F1-score for all models in the Large-Scale Evaluation for the Enhanced Prompt

	Gemini 2.5 Flash		GPT-OSS 120b		GPT-OSS 20b	
	Acc	F1	Acc	F1	Acc	F1
Total All Topics	0.72	0.71	0.69	0.71	0.66	0.68
Expressions	0.80	0.80	0.90	0.90	0.85	0.85
Logical Expr.	0.73	0.70	0.80	0.81	0.80	0.80
Variables	0.68	0.66	0.61	0.62	0.56	0.58
Conditionals	0.71	0.72	0.61	0.64	0.63	0.66
Loops	0.74	0.71	0.72	0.73	0.68	0.69
Functions	0.45	0.41	0.76	0.76	0.71	0.71
Pointers	0.88	0.87	0.70	0.77	0.58	0.62
Arrays	0.73	0.72	0.56	0.60	0.58	0.60

4.4 Using Fine-Tuning for Improving Model Accuracy

This section explores the third iteration of the diagnostic component. The architecture defined previously demonstrated that large commercial models, such as Gemini 2.5-Flash, have robust diagnostic capabilities and an adequate identification of the skills and cognitive stage defined in the learning framework; in addition, those models perform generally better than open-weight models. However, commercial LLMs present significant challenges in terms of cost, limited free tiers, and data privacy, which represent a challenge for a final real-time system. Consequently, it becomes necessary to rely on an open-weights model architecture that can be run locally.

To achieve this, the GPT-OSS-20b model was selected due to its optimal balance between computational efficiency and advanced reasoning capacity, which makes it highly suitable to run in a cluster environment. However, this base open-weight model exhibits limited accuracy in the task of identifying missing skills in the learning framework. For that reason, a fine-tuning strategy is used to increase its accuracy, trying to get values similar to or even better than the Gemini 2.5-Flash model. This strategy follows a well-defined three-stage process:

- **Dataset Construction:** Use the Moodle Logs of the C Programming course to create a dataset of real student information.
- **Knowledge Distillation:** Transfer the reasoning capabilities of a "Teacher Model" into a smaller "Student Model", specifically using a powerful model like Claude Sonnet 4.5

to generate high-quality labels to complete the dataset according to the learning framework.

- **Fine-Tuning:** Training the Student model on this labeled dataset to achieve better accuracy, evaluating its performance in identifying the missing skills task.

The general process following this architecture is summarized in Figure 4.23.

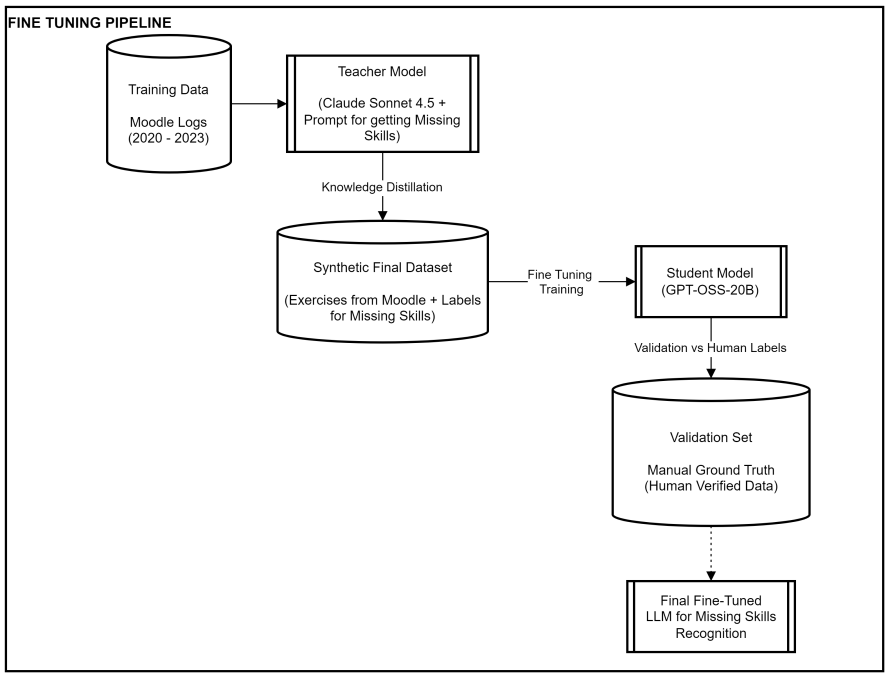

Figure 4.23: Fine-Tuning Process for the *Diagnostic Engine*.

4.4.1 Dataset Construction & Knowledge Distillation

Phase 1 - Dataset Construction: For the fine-tuning task, a dataset is needed for training. In this case, we began by analyzing the Moodle Logs of the Programming Course for the bachelor's degree in Computer Science at the University of Padua, which were provided by the faculty. These logs have information on homework, VPL (Virtual Programming Laboratories) assignments, and some tests taken from previous years (2020-2021, 2021-2022, 2022-2023, 2023-2024). In order to have a strong dataset, we analyzed logs from years: 2020-2021, 2021-2022, 2022-2023. First, a Python script was created to process the XML files into readable Excel files to identify the exercises that are useful for the task; i.e., the exercises were categorized

by the topic each is treating (refer to the topics list in Section 3.3). The output of this first task is a list of exercises categorized by topic and metadata information such as title, general description, and base code. After that, for each exercise, all student submissions were considered part of the dataset.
In addition, a critical step was to validate these new datasets against the Ground Truth dataset used for testing the models. This is because both datasets use part of the same Moodle logs. For that reason, we implemented a filter to strictly exclude any submission that was already part of the testing ground truth; this specification matches the same exercise id and the same student id to identify a duplication in the datasets. This process ensures that the fine-tuning model will never be trained on the same data used to validate it.
Secondly, the next step of this dataset construction was to manually identify the skills each exercise was evaluating to add them to the exercise information. This is because for the fine tuning process, it will be necessary to teach the open-weights model which are the correct answers for the learning framework.
Finally, the complete dataset ends with a total of 36 exercises and 3648 student submissions, which represents a dataset of exercises with answers from a real programming course, but without learning framework labels.

Phase 2 - Knowledge Distillation: In this step, we have a raw dataset ready with exercises, real students' answers, and their corresponding grades. Now the challenge is that these 3,648 submissions lacked specific labels for the learning framework (Missing Skills, Mastered skills, and Cognitive Stage), which make a manual labeling unfeasible; for that reason, the Knowledge Distillation strategy was employed to obtain automatic labels.
As described by Shirgaonkar et al. [26], Knowledge Distillation is a process that can significantly improve the performance of smaller models with the help of teacher-generated high-quality synthetic data. Based on this information, it was necessary to select a "Teacher model" capable of creating high-quality labels for the raw dataset; the selected model was Claude Sonnet 4.5 due to its good performance during the qualitative analysis of the baseline prompt shown in Section 4.2.2.
In order to automate the labeling process, a Python script was created to read the raw dataset and create batches of 100 exercises each. For each batch, the code snippet uses the same prompt as in section 4.3.1 to ask the model to generate as output a JSON structure containing the appropriate labels: missing skills, mastered skills, and cognitive stage. In addition, another field was also asked to the "Teacher Model"; it was the analysis section about why the code snippet

has those missing skills; this was with the aim of using that information for the fine-tuning process. In this way, Claude Sonnet 4.5 is exposed to the same tested prompt for analyzing the missing skills, in which the Neo-Piaget Stages model is injected to let the model analyze the skills according to the learning framework.
The final step is to join the answers generated by the "Teacher Model" (Claude Sonnet 4.5) to the existing raw dataset; this is possible because we have the key: exercise_id and student_id. The result is a fully labeled dataset in which every student submission now has a detailed cognitive diagnosis. This dataset is then ready to be fed into the "Student Model" (GPT-OSS-20b), which will run locally.

4.4.2 Baseline Fine-Tuning Architecture

This is the initial iteration of phase 3: fine-tuning training. In this section, the model was trained using the final dataset produced in the previous section. This was then transformed into a JSONL format due to its enhanced performance with the Hugging Face library, which, instead of reading all the data in memory, reads them line by line to prevent out-of-memory errors. This JSONL dataset has one line per each code submission with the following information, an example of one of the values of the dataset is shown in the algorithm 4.1:

- Instruction, which refers to the prompt used previously.
- Chain of Thought is the analysis of missing skills.
- Input is the student's submission (C code).
- Output, which represents the labels for that specific submission (missing skills, mastered skills, and cognitive stage).

The GPT-OSS-20b model was then fine-tuned using a parameter-efficient fine-tuning (PEFT) approach based on Low-Rank Adaptation (LoRA). In this strategy, the base model was loaded in half-precision (BF16) with numerical stability, while only the LoRA adapters were updated during the training process, keeping the original weights of the base model frozen. This process was performed using the cluster of the Department of Mathematics of the University of Padua, where the job was run on a node with an Nvidia RTX A6000 GPU with 48 GB of VRAM. Although a Quantized LoRA was initially considered, it could not be used due to compatibility limitations between a 4-bit quantization and the GPT-OSS-20b model. This was caused because the base model is a MoE (Mixture of Experts) model, with a combination of weights stored in 4-bit MXFP4 format, which requires a special infrastructure [33].

Algorithm 4.1 Structure of an instance of the dataset for the Fine Tuning Process

Initialize a `messages` array for a single interaction

// Turn 1: System Instructions
Set Role: `system`
Set Content: "You are an expert C programming teacher..."
  Define Cognitive Model (Piaget Stages)
  Define Skills List (S1, S2, P1, C1, etc.)
  Enforce strict JSON output format

// Turn 2: Student Submission
Set Role: `user`
Set Content: Raw student code string
  `Example: "#include <stdio.h> ... int main() { ... }"`

// Turn 3: Teacher Model Diagnosis (Ground Truth)
Set Role: `assistant`
Set Thinking: Chain of Thought reasoning process
  `Example: "The student correctly implements integer arithmetic..."`
Set Content: Final parsed JSON string
  `mastered_skills:` ["C1", "C2", "P1", "P2", ...]
  `missing_skills:` ["None"]
  `cognitive_stage:` "Concrete-operational"

To perform the fine-tuning training of the model, some specific libraries were needed. It is important to mention that, as GPT-OSS-20b is a relatively new model, some libraries require specific versions to recognize the structure of the model. In that way, two conda environments were created: one for the training process with a special requirement of Transformers v.4.55.0 library, and the second environment for the inference with the library vllm v.0.13.0. The specific versions for both environments are:

- Accelerate (1.12.0)
- BitesAndBytes (0.49.0)
- HuggingFace-hub (0.36.0)
- PEFT (0.18.0)
- Transformers (4.55.0)

- Torch (2.9.1)
- Tokenizers (0.21.4)
- Triton (3.5.1)
- Vllm (0.13.0)

Before continuing with the fine-tuning training, one distinctive aspect of the GPT-OSS model is its use of the Harmony prompt format [33]. This format structures conversations into a sequence of messages with different roles, such as system, user, assistant, and tools; and it explicitly separates the assistant's reasoning process from its final answer; i.e., the model supports the use of a chain of thought as part of the assistant role. For that reason, it is essential to format the dataset we created in this Harmony style for the GPT-OSS model to work correctly; otherwise, the inference of the fine-tuned model substantially decreases. In general, the harmony template defines the mentioned roles as follows [33]:

- **System:** instructions for the model (prompt).
- **User:** message from the final user.
- **Assistant:** answer provided by the model. It also provides a special analysis channel for the model thoughts.
- **Tool:** outputs from external tools (not used in this case).

As described above, the tagged dataset created in the previous step must be processed in the Harmony template to align with the GPT-OSS-20b model. In that way, algorithm 4.2 shows an example of a formatted student submission part of the final dataset.
Once the environments were set up and the dataset was formatted, the next step was to split the current dataset into training and test datasets, with a configuration of 80% and 20%. Fine-tuning was then performed with $\gamma = 8, \alpha = 16$, LoRA dropout of 0.05, and the training layers were *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj* and *down_proj*. The model was trained for one epoch using the Adam optimizer with a learning rate of $1e-5$, saving evaluations every 100 steps. With this configuration, only 0.019% of the model parameters were updated, resulting in approximately 4 millions of parameters. After the training process, the result was the adapters of the model, which can be used directly with LoRA configuration or merged with the base model.

Algorithm 4.2 Harmony Template for Fine-Tuning Training in GPT-OSS-20b model.

```
{
  "messages": [
    {
      "role": "system",
      "content": "You are an Automated Diagnostic System for C programming..."
    },
    {
      "role": "user",
      "content": "Student Code:int main() { int x = 0;
         while(x < 5) { printf(\"%d\\n\", x); } }"
    },
    {
      "role": "assistant",
      "content": "{\"cognitive_stage\": \"Pre-operational\",
         \"missing_skills\": [\"P2\"],
         \"mastered_skills\": [\"S1\", \"C1\"]}"
    }
  ]
}
```

The next step was to evaluate the performance of the fine-tuned model using the vllm library for inference, which provides a fast inference service for the model. The aim of this evaluation is to determine what impact it has on the accuracy diagnosis the use of this baseline fine-tuned model with specific Neo-Piaget labels in an open-weight model trained using historical Moodle logs.

In that way, this experiment starts using the enhanced prompt defined in the previous sections. Regarding the training metrics, after one epoch, the model achieved a training loss of 0.3791 and an evaluation loss of 0.4264. These values indicate a stable learning progression and suggest that there are no likely signs of model overfitting; Figure 4.24 shows the corresponding loss curve.

As in previous experiments, the evaluation of the fine-tuned model was performed using the same methodology presented in Section 4.1. This experiment achieves a total accuracy of 67,64% and a weighted F1 score of 68.11%, which represents an improvement with respect to the base model, which achieved an accuracy of 65.94% obtained in the last test. Analyzing

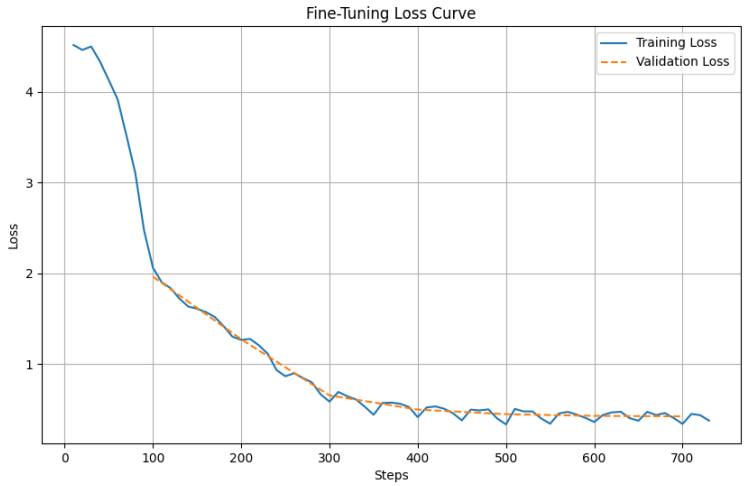

Figure 4.24: Loss Curve for Fine-Tuning process.

each topic, the evaluation of the "Loops" topic revealed the best accuracy with 76.39%, these results are shown in Table 4.20.

Table 4.20: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.6764	0.6811
Expressions	0.7000	0.7444
Logical Expressions	0.7250	0.7293
Variables	0.6585	0.6522
Conditionals	0.7073	0.7117
Loops	0.7639	0.7678
Functions	0.6316	0.5803
Pointers	0.7500	0.7566
Arrays	0.5657	0.5963

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model. Table 4.21 shows the metrics for each topic, considering the analysis of the missing and mastered skills identified.

This analysis shows that almost all the precision values are high, except for the identification of missing skills of the topic Arrays. However, the recall values can be improved because some of them have very low values; for instance, the identification of missing skills in the topic "Ex-

Table 4.21: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.671	0.733	0.208	0.324
MA-Expressions	0.639	0.661	0.862	0.748
MI-Logical Expressions	0.761	0.759	0.308	0.439
MA-Logical Expressions	0.777	0.774	0.961	0.858
MI-Variables	0.798	0.660	0.324	0.435
MA-Variables	0.803	0.809	0.940	0.869
MI-Conditionals	0.812	0.790	0.405	0.536
MA-Conditionals	0.800	0.758	0.885	0.817
MI-Loops	0.847	0.625	0.246	0.353
MA-Loops	0.822	0.824	0.923	0.871
MI-Functions	0.752	0.795	0.200	0.320
MA-Functions	0.746	0.711	0.962	0.817
MI-Pointers	0.744	0.810	0.228	0.356
MA-Pointers	0.754	0.752	0.951	0.840
MI-Arrays	0.833	0.428	0.330	0.373
MA-Arrays	0.818	0.756	0.807	0.781

pressions" and "Logical Expressions" that do not exceed the 30%. As shown in Figure 4.25, the confusion matrix of the results shows that the model does a good job recognizing the Pre-operational and Concrete Operational stages, but it failed 50 times in recognizing the Concrete Operational Stage. In addition, during the evaluation process, the model made some relevant errors. For instance, the model identified a code snippet with all the skills as mastered, but it categorized it as the Sensorimotor Stage. In other cases, the model does not identify any missing or mastered skill but writes the cognitive Stage as Concrete Operational.

These failures let us conclude that the fine-tuned model could achieve a slightly higher accuracy value compared to the base model, but it still has relevant errors when it follows the defined learning framework. For that reason, it is necessary to not treat the model as just a classifier for C programming; instead, it must keep its reasoning capabilities, which means that the fine-tuning training should teach the model how to think and identify the missing skills correctly.

4.4.3 Fine-Tuning Architecture using Chain of Thought (CoT)

In this new strategy, the dataset will include the thoughts of the Teacher Model (refer to phase 2 - Knowledge Distillation) as part of the assistant's thinking role of the Harmony template

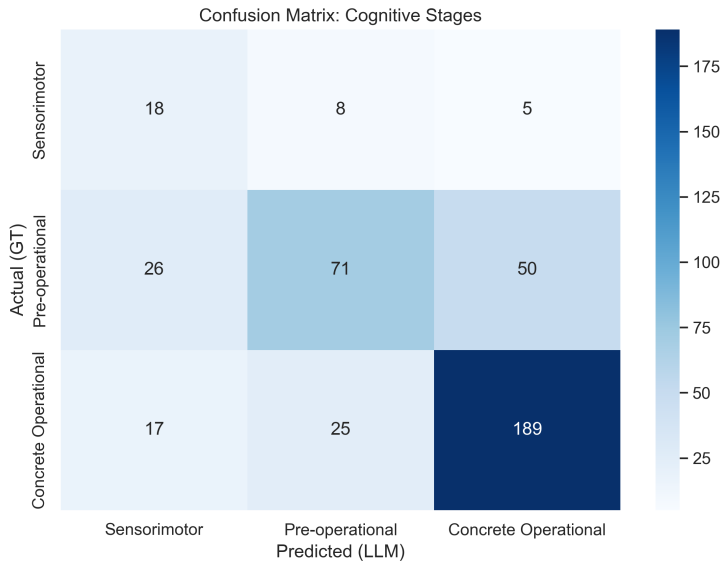

Figure 4.25: Confusion Matrix in Fine-Tuned GPT-OSS-20b model.

described above. This is done with the aim of teaching the model why some skills are missing in specific code snippets.

In this case, again, the dataset is first transformed into JSONL format for processing. Similarly, the current dataset was divided into training and test datasets, with a configuration of 80% and 20%. The parameters used in this fine-tuning training were: $\gamma = 8$, $\alpha = 16$, LoRA dropout of 0.05, and the training layers were *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj* and *down_proj*. The model was trained for 1.5 epochs using the Adam optimizer with a learning rate of $2e-5$, saving evaluations every 100 steps. With this configuration, only 0.019% of the model parameters were updated, resulting in approximately 4 millions of parameters.

After 1.5 epochs, the model achieved a training loss of 0.2251 and an evaluation loss of 0.2598, again showing no signs of model overfitting; figure 4.26 shows the loss curve.

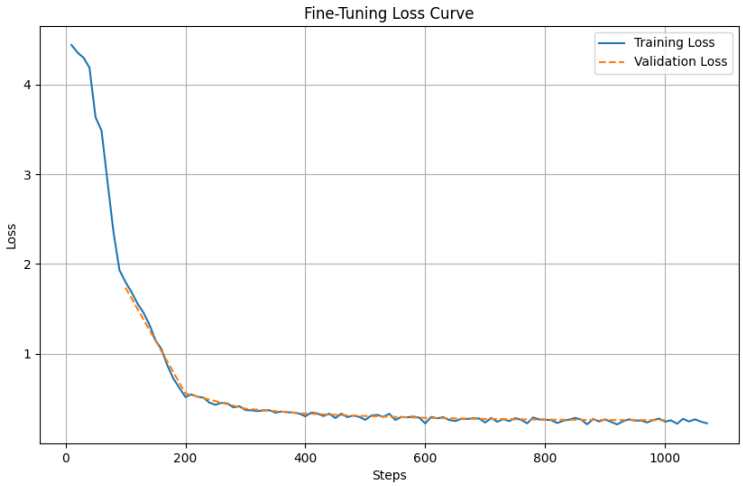

Figure 4.26: Loss Curve for Fine-Tuning process using CoT.

After the training process, the inference process was performed using the vllm library as in the previous experiment. Subsequently, the evaluation of the fine-tuned model was performed using the same methodology as in Section 4.1. This experiment achieves a total accuracy of 68.86% and a weighted F1 score of 65.25%, which represents an improvement over the previous attempt, which achieved an accuracy of 67.64%. Analyzing each topic, the evaluation of the "Logical Expressions" topic revealed the best accuracy with 80%, these results are shown in Table 4.22.

Table 4.22: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.6886	0.6525
Expressions	0.7000	0.6965
Logical Expressions	0.8000	0.7810
Variables	0.6341	0.5741
Conditionals	0.7317	0.7223
Loops	0.6528	0.5778
Functions	0.7105	0.6976
Pointers	0.7250	0.6980
Arrays	0.6465	0.5902

The multi-label analysis of individual skills identifies the strengths and weaknesses of the model.

Table 4.23 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

Table 4.23: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.754	0.761	0.509	0.610
MA-Expressions	0.846	0.907	0.839	0.872
MI-Logical Expressions	0.816	0.964	0.406	0.571
MA-Logical Expressions	0.820	0.802	0.987	0.885
MI-Variables	0.834	0.811	0.398	0.534
MA-Variables	0.838	0.836	0.956	0.892
MI-Conditionals	0.814	0.761	0.446	0.562
MA-Conditionals	0.798	0.764	0.868	0.812
MI-Loops	0.839	0.537	0.352	0.426
MA-Loops	0.856	0.884	0.895	0.889
MI-Functions	0.782	0.767	0.361	0.491
MA-Functions	0.769	0.752	0.908	0.823
MI-Pointers	0.769	0.788	0.349	0.484
MA-Pointers	0.779	0.773	0.954	0.854
MI-Arrays	0.872	0.582	0.525	0.552
MA-Arrays	0.864	0.801	0.881	0.839

This analysis shows that all the precision values are high, except for the identification of missing skills of the topic Arrays. However, the recall values can be improved because some of them have very low values; for instance, the identification of missing skills in the topic "Expressions" and "Logical Expressions" that do not exceed the 30%, which means that the model is skipping to make correct validations on missing skills. In addition, with these results, we can conclude that it is easier for the model to detect the mastered skills than the missing skills. As shown in Figure 4.27, the confusion matrix of the results shows that the model does a good job recognizing the Pre-operational stage. However, the model tends to predict more exercises as the Pre-operational stage instead of the Concrete Operational stage.

In conclusion, besides the Chain of Thought technique increasing the general accuracy of the fine-tuned model, it does not outperform the reasoning capabilities of commercial models like Gemini-2.5-Flash. In this scenario, the fine-tuning process has not yet yielded the expected results because although it has improved the accuracy of the base model, it is still not fully comparable to the Gemini model.

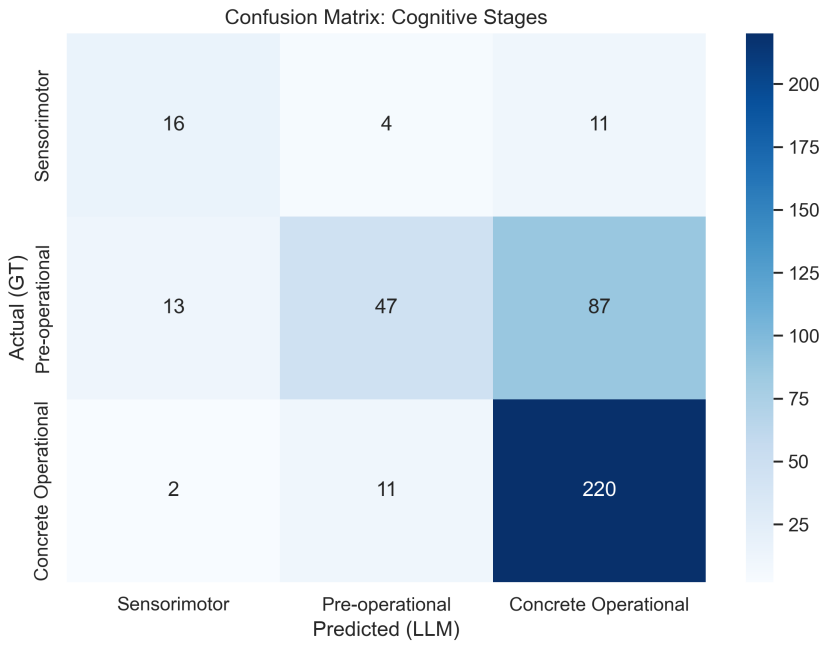

Figure 4.27: Confusion Matrix in Fine-Tuned GPT-OSS-20b model with CoT.

4.4.4 Fine-Tuning Attempt Using More Aggressive Training Parameters

Taking into account the relative improvement of the last architecture, in this experiment, the same Chain of Thought strategy was used, but this time it was trying to implement a more aggressive hyperparameter tuning during the training process. The parameters used in this fine-tuning training were: $\gamma = 16$, $\alpha = 32$, LoRA dropout of 0.05, and the training layers were *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj* and *down_proj*. The model was trained for 3 epochs using the Adam optimizer for 32-bit precision with a learning rate of $5e-5$, saving evaluations every 100 steps and using a value of 8 steps for gradient accumulation. With this configuration, the number of updated parameters increased to 0.038%, resulting in approximately 8 million parameters, i.e., twice the number of parameters compared to the last attempt.
After 3 epochs, the model achieved a training loss of 0.1094 and an evaluation loss of 0.1094, which represents a more stable model learning, with no signs of overfitting; figure 4.28 shows the loss curve.
After the training section, the inference process was performed using the vllm library as in the previous experiment. In the same way, the evaluation of the fine-tuned model was performed

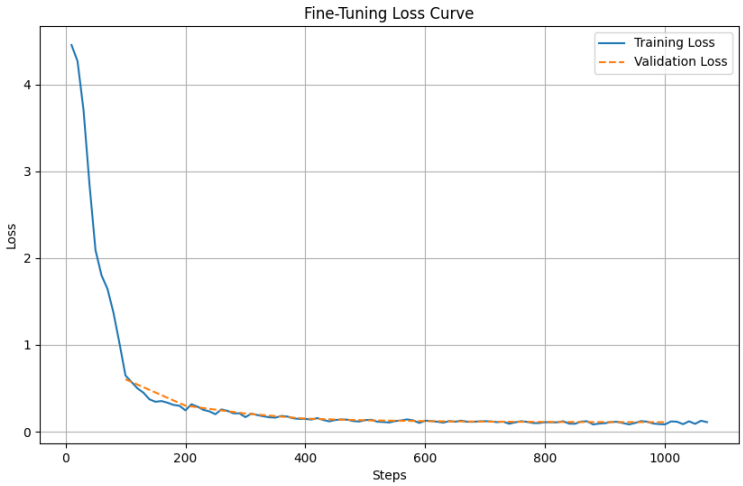

Figure 4.28: Loss Curve for Fine-Tuning process using CoT and a more aggressive parameter configuration.

using the same methodology as in Section 4.1. This experiment achieves a total accuracy of 70.32% and a weighted F1 score of 70.22%, which represents an improvement over the previous attempt, which achieved an accuracy of 68.86%. Analyzing each topic, the evaluation of the "Logical Expressions" and "Pointers" topics revealed the best accuracy with 80%, these results are shown in Table 4.24.

Table 4.24: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT and a more aggressive parameter configuration - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.7032	0.7022
Expressions	0.7750	0.7720
Logical Expressions	0.8000	0.8137
Variables	0.7805	0.7635
Conditionals	0.7317	0.7534
Loops	0.7083	0.6746
Functions	0.6316	0.6052
Pointers	0.8000	0.8106
Arrays	0.5758	0.5941

Table 4.25 shows the metrics for each topic, considering the analysis of the missing and mastered skills identified.

Table 4.25: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model with CoT and a more aggressive parameter configuration - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.736	0.758	0.443	0.560
MA-Expressions	0.736	0.729	0.914	0.811
MI-Logical Expressions	0.800	0.800	0.451	0.577
MA-Logical Expressions	0.800	0.800	0.951	0.869
MI-Variables	0.845	0.797	0.472	0.593
MA-Variables	0.849	0.848	0.956	0.899
MI-Conditionals	0.785	0.636	0.463	0.536
MA-Conditionals	0.769	0.757	0.797	0.777
MI-Loops	0.861	0.667	0.361	0.468
MA-Loops	0.831	0.798	0.989	0.883
MI-Functions	0.825	0.970	0.413	0.579
MA-Functions	0.791	0.739	1.000	0.850
MI-Pointers	0.733	0.648	0.309	0.418
MA-Pointers	0.733	0.741	0.932	0.826
MI-Arrays	0.849	0.499	0.677	0.574
MA-Arrays	0.849	0.840	0.771	0.804

Similarly to the last attempt, in this case, almost all the precision values are high, except for the identification of the missing skills of the "Arrays" topic, which has a reduction compared to the last experiment. In contrast, the recall values are slightly higher compared to the last results in Table 4.23. However, there are still recall values with less than 50%; for instance, the recognition of the missing skills in the "Pointers" topic. In addition, as shown in Figure 4.29, the confusion matrix of the results shows that the model is doing a better job of correctly recognizing between Concrete Operation and Pre-operational stages. However, the Sensorimotor stage is demonstrating a problem because the model tends to predict more Pre-operational stages instead of the Sensorimotor stage.

In conclusion, the combination of the use of Chain of Thought in fine-tuning training and the use of more aggressive hyper-parameters resulted in a better performance of the model in its task of detecting the learning programming framework. Even if a 70.32% accuracy is an improvement of the base model, which obtained only 65.94%, it does not outperform the capabilities of more powerful models like Gemini-2.5-Flash. Another important point is the difference between the precision and recall values in the identification of missing skills, which lets us conclude that the model exhibits a more conservative diagnostic behavior, trying to prioritize high precision in Missing Skills in order to minimize false positives. However, this makes the model

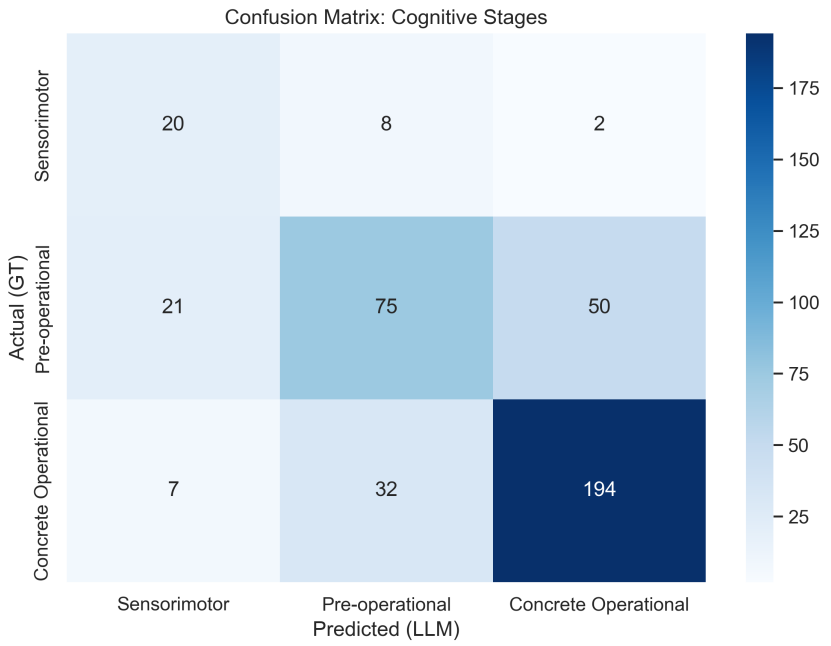

Figure 4.29: Confusion Matrix in Fine-Tuned GPT-OSS-20b model with CoT and a more aggressive parameter configuration.

forget other important skills because it prefers not to mark skills as errors if the model is not completely sure of them. In contrast, the diagnosis of mastered skills exhibits a different scenario, since in this case the precision and recall values are both high. This means that for the model, it is easier to detect what is correct in the programming code than to know if there is a syntax or logical error.

With regard to these results, it is easy to see that the model is missing some reasoning capabilities. It could be identified in the thinking steps the model performs before answering, which are short and do not specify the evaluation of all the skills; an example is shown in Figure 4.30. This is an indicator of some points of improvement on the dataset with which the model was fine-tuned; for instance, it is necessary that the dataset has more relevant information on the thinking label, so the fine-tuned model could learn how to think to generate a diagnosis. In addition, as presented in the methodology section, the actual dataset has an imbalance between the number of examples for each cognitive stage. As the majority of students did a good job during the Moodle labs, their logs have more examples of students' submissions with correct implementations. For that reason, the model is struggling to detect those examples. Finally, dealing with the dataset imbalance problem is a relevant point of improvement for the fine-

tuning of the GPT-OSS-20b model.

```
[2026-01-08 22:25:04]
Ex: ARR_04 | Stu: 1343
CODE:
#include<stdio.h>int main(){ int N; int A[N]; int ppal=1; //leggere da input grandezza array scanf("%d", &N); // ... for(int i = 0;
[2026-01-08 22:25:04] Assistant:
--- REASONING ---
Missing P1 because reading into uninitialized array length leads to UB.

--- JSON ---
{"mastered_skills":["C1","C7"],"missing_skills":["P1","P7"],"cognitive_stage":"Pre-operational"}

--------------------------------------------------------------------------------
[2026-01-08 22:24:53]
Ex: ARR_03 | Stu: 2318
CODE:
#include <stdio.h>int main(void){ int array[] = {3, 5, 1, 7, 0, 9, 4, 6, 2, 8}; int min_indice = 0, max_indice = 0; int dim; dim = s
[2026-01-08 22:24:53] Assistant:
--- REASONING ---
Missing P1 because loop accesses array[i+1].
The student fails to trace correctly due to boundary violation.
Correct reasoning.

--- JSON ---
{"mastered_skills":["C1","C7"],"missing_skills":["P1"],"cognitive_stage":"Pre-operational"}

--------------------------------------------------------------------------------
[2026-01-08 22:24:49]
Ex: ARR_03 | Stu: 2487
CODE:
#include <stdio.h>/*...*/int main (void) {int scatola[10]= {3, 5, 1, 7, 0, 9, 4, 6, 2, 8};int minimo= 10000;int massimo= -10000;int
[2026-01-08 22:24:49] Assistant:
--- REASONING ---
The solution correctly implements bounds checking and tracks indices. It demonstrates mastery of loop logic and state management.
All required skills present.
No further issues.

--- JSON ---
{"mastered_skills":["C1","C7","P1","P7","S1","S2","S3"],"missing_skills":["None"],"cognitive_stage":"Concrete-operational"}
```

Figure 4.30: Reasoning step of the Fine-tuned model using CoT training.

In conclusion, the evaluation of this strategy revealed that the model still struggled with reasoning depth and faced challenges due to dataset class imbalance, as it does not know how to act for errors in the Sensorimotor and Pre-operational stages. This is because the majority of students did a good job during the Moodle labs, their logs have more examples of students' submissions with correct implementations. Tables 4.26 and 4.27 shows a comparison of the total number of exercises divided by categories; as can be seen, the number of Sensorimotor and Pre-operational stages is much lower than the number of submissions categorized as the Concrete Operational stage.

4.4.5 Fine-Tuning Attempt Using an Augmented Dataset

As presented above, the imbalance dataset is the main improvement the model needs in order to outperform general purpose models. For that reason, it is fundamental to increase the number of code submission examples to obtain a more balanced dataset for each stage.

Table 4.26: Total Number of Submissions by Cognitive Stage of the dataset for the Fine-tuning process.

Cognitive Stage	Number of Submissions
Sensorimotor	108
Pre-operational	726
Concrete Operational	2753

Table 4.27: Number of Submissions by Cognitive Stage for each topic of the dataset for the Fine-tuning process.

Topic	Stage	Count	Percentage
Variables	Concrete-operational	113	74.83%
Variables	Sensorimotor	4	2.65%
Variables	Pre-operational	34	22.52%
Conditionals	Concrete-operational	161	61.92%
Conditionals	Pre-operational	99	38.08%
Loops	Concrete-operational	772	80.75%
Loops	Pre-operational	166	17.36%
Loops	Sensorimotor	18	1.88%
Functions	Concrete-operational	259	87.21%
Functions	Sensorimotor	16	5.39%
Functions	Pre-operational	22	7.41%
Arrays	Concrete-operational	1448	75.30%
Arrays	Pre-operational	405	21.06%
Arrays	Sensorimotor	70	3.64%

In order to do the augmentation, it is not possible to get real information about code submissions from Moodle Logs because real students' interactions do not have basic errors in the Sensorimotor or Pre-operational stages, especially during the early topics. Consequently, the augmentation process needs to be done using artificial examples generated from an LLM with high reasoning capabilities; in this case, the selected model was Claude Sonnet 4.5.

The general purpose of augmenting the dataset with artificial code submissions is to get examples with specific errors in different topics; however, asking the LLM to generate examples with any type of error does not guaranty that the model will generate the correct error, depending on the specific stage. For that reason, a list of common errors defined by topic and stage was used, using as a reference Appendix A presented by Sorva et al. [2], in which the authors collect a list of common misconceptions that students have when learning introductory programming topics. This list of types of error is presented in Table 4.28, in which, depending on the topic,

a few specific misunderstandings were selected and classified as errors in the Sensorimotor or Pre-operational stage.

Table 4.28: Types of errors observed in generated code submissions for the augmented dataset during the fine-tuning process.

Topic	Type of Error	Cognitive Stage
Expressions	Limited understanding of expressions, lacking the concept of evaluation.	Sensorimotor
Expressions	Difference in data types (e.g., confusion between `int` and `float`).	Pre-operational
Expressions	Misinterpretation of variable declaration order affecting value assignment.	Sensorimotor
Variables	Difficulty understanding the lifetime and scope of values.	Pre-operational
Variables	Belief that variables can hold multiple values simultaneously.	Sensorimotor
Variables	Syntax-related errors.	Sensorimotor
Variables	Logical errors involving operators.	Sensorimotor
Conditionals	Difficulties understanding the sequential execution of statements.	Sensorimotor
Conditionals	Belief that a false condition terminates the program when no `else` branch is provided.	Pre-operational
Conditionals	Syntax-related errors.	Sensorimotor
Loops	Difficulties understanding automatic updates of loop control variables.	Pre-operational
Loops	Misconception that `while` loops terminate immediately once the condition becomes false.	Pre-operational
Loops	Belief that return values do not need to be stored, even outside loops.	Pre-operational
Functions	Sensorimotor-level understanding of value flow between functions.	Sensorimotor
Functions	Difficulties understanding method invocation from other methods.	Pre-operational
Arrays	Confusion between an array and its individual elements.	Pre-operational

Topic	Type of Error	Cognitive Stage
Arrays	Difficulties handling subscripts and indices in two-dimensional arrays.	Pre-operational
Arrays	Difficulties with arrays that store indices as data.	Pre-operational

Using the list of errors presented, a new prompt was developed in Figure 4.31 in order to ask the LLM (Claude Sonnet 4.5) to generate a specific number of responses for each of the exercises in the original dataset. In this prompt, depending on the exercise, the types of errors were passed as an argument.

Prompt for generating new code submissions for the Fine-tuned augmented dataset.

System Role: You are an expert C programming tutor for the topic Arrays who acts as an Automated Exercise Generator. Your goal is to generate exercises based on specific errors using the Piaget cognitive model presented below.

Cognitive Model (Piaget Model):
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how program flow works.
2. **Pre-operational** – Students trace and interpret the code's execution but struggle with abstraction.
3. **Concrete Operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Instructions:
- Taking into account the cognitive model above, generate 60 code answers for the exercise below.
- The answers must contain mistakes for the Sensorimotor or Pre-operational stages.
- 30 for Sensorimotor stage.
- 30 for Pre-operational stage.
- The mistakes must be of the following types depending on the stage:

Sensorimotor Errors:
- Syntax errors.
- Confusion between an array and its cell.

Pre-operational Errors:
- Difficulties with dealing with 2D array subscripts and dimensions.
- Difficulties with arrays containing indices as data.

Output Format: The output must be a JSON object containing the answers using the following format.

```
 "ARR_21": {
"submissions": [
{
"stage": "Sensorimotor | Pre-operational",
"code": "answer code",
"error_type": "error type"
}   ] }
```

Exercise:

Figure 4.31: Prompt for generating new code submissions for the Fine-tuned augmented dataset.

After this process, a new dataset was created with a total of 5 698 submissions, which represents an increase of 35.97% with respect to the original dataset, which has a total of 3 648 submissions. The augmented examples were manually reviewed to ensure their consistency with the Neo-Piaget framework. Tables 4.29 and 4.30 show a comparison of the total number of exercises divided by categories; as can be seen, the percentage of Sensorimotor and Pre-operational examples increased significantly, to around 20% and 40% respectively. This represents a more stable distribution of cognitive stages in the dataset. In addition to each of these new examples, the tags of missing and mastered skills were added using the same LLM (Claude Sonnet 4.5), including the reasoning steps the model used to get the list of skills.
Once the dataset was complete, the training process was performed using the same Chain of Thought strategy, but this time it was completed with more detailed reasoning information in order to teach the fine-tuned model why some skills are missing and others are mastered. The parameters used in this fine-tuning training were the same as in the previous experiment: $\gamma = 16$, $\alpha = 32$, LoRA dropout of 0.05, and the training layers were *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj*, and *down_proj*. The model was trained again for 2 epochs using the Adam

Table 4.29: Total Number of Submissions by Cognitive Stage of the Augmented dataset for the Fine-tuning process.

Cognitive Stage	Number of Submissions
Sensorimotor	1083
Pre-operational	1862
Concrete Operational	2753

Table 4.30: Number of submissions by cognitive stage for each topic in the augmented dataset used for the fine-tuning process.

Topic	**Cognitive Stage**	**Count**	**Percentage**
Variables	Concrete-operational	113	37.92%
	Sensorimotor	77	25.84%
	Pre-operational	108	36.24%
Conditionals	Concrete-operational	161	37.01%
	Pre-operational	214	49.20%
	Sensorimotor	60	13.79%
Loops	Concrete-operational	772	54.71%
	Pre-operational	461	32.67%
	Sensorimotor	178	12.62%
Functions	Concrete-operational	259	62.11%
	Sensorimotor	76	18.23%
	Pre-operational	82	19.66%
Arrays	Concrete-operational	1448	46.16%
	Pre-operational	997	31.78%
	Sensorimotor	692	22.06%

optimizer for 32-bit precision with a learning rate of $5e-5$, saving evaluations every 100 steps and using a value of 8 steps for gradient accumulation. With this configuration, the number of updated parameters increased to 0.038%, resulting in approximately 8 million parameters. After 2 epochs, the model achieved a training loss of 0.085 and an evaluation loss of 0.096, which represents a more stable model learning, with no signs of overfitting; figure 4.32 shows the loss curve.

After the training section, the inference process was performed using the vllm library as in the previous experiments. In the same way, the evaluation of the fine-tuned model was performed using the same methodology as in Section 4.1. This experiment achieves a total accuracy of 73.72% and a weighted F1 score of 75.04%, which represents a significant improvement over the

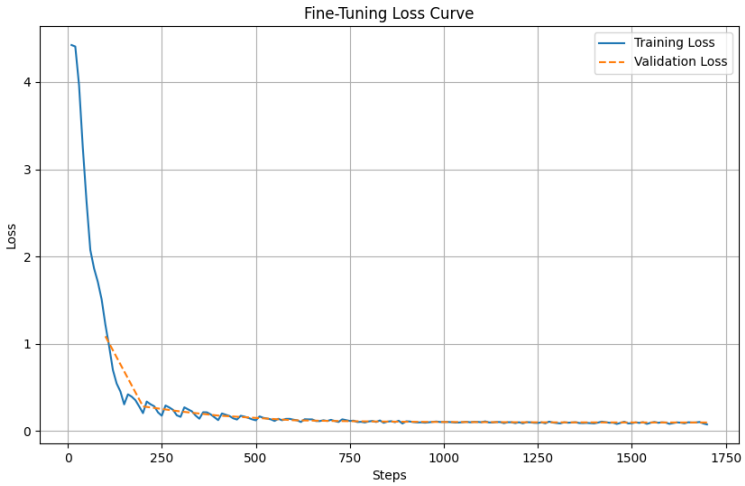

Figure 4.32: Loss Curve for Fine-Tuning process using the augmented dataset.

previous attempt, which achieved an accuracy of 70.32%. Analyzing each topic, the evaluation of the "Loops" topics revealed the best accuracy with 79.16%, these results are shown in Table 4.31.

Table 4.31: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model using the augmented dataset - Metrics by Topic

Topic	Accuracy	F1-Weighted
Total All Topics	0.7372	0.7504
Expressions	0.7500	0.7904
Logical Expressions	0.7500	0.7747
Variables	0.7317	0.7545
Conditionals	0.7317	0.7503
Loops	0.7917	0.7905
Functions	0.7368	0.7247
Pointers	0.7000	0.7433
Arrays	0.7071	0.7308

Table 4.32 shows the metrics for each topic, considering the analysis of identified missing and mastered skills.

In this experiment, we can notice that almost all skills have a higher accuracy, around 70%, which is an improvement over the last experiment. Regarding the precision, there are some values that are not high; for example, the identification of the missing skills of the "Loops"

Table 4.32: Quantitative Evaluation for Fine-Tuned GPT-OSS-20b Model using the augmented dataset - Metrics by Skill (MI: Missing Skills, MA: Mastered Skills)

Topic	Accuracy	Precision	Recall	F1
MI-Expressions	0.814	0.755	0.755	0.755
MA-Expressions	0.814	0.851	0.851	0.851
MI-Logical Expressions	0.834	0.711	0.759	0.735
MA-Logical Expressions	0.834	0.893	0.866	0.879
MI-Variables	0.874	0.695	0.843	0.762
MA-Variables	0.878	0.948	0.873	0.909
MI-Conditionals	0.867	0.740	0.777	0.758
MA-Conditionals	0.865	0.907	0.815	0.858
MI-Loops	0.864	0.587	0.664	0.623
MA-Loops	0.864	0.909	0.878	0.893
MI-Functions	0.765	0.703	0.335	0.454
MA-Functions	0.767	0.735	0.946	0.827
MI-Pointers	0.787	0.658	0.658	0.658
MA-Pointers	0.796	0.843	0.858	0.851
MI-Arrays	0.871	0.548	0.795	0.648
MA-Arrays	0.876	0.896	0.783	0.836

and ”Arrays” topic, which has a value around 55%. In contrast, the recall values are slightly higher compared to the last results in Table 4.25. However, there are still recall values with less than 50%; for instance, the recognition of the missing skills in the ”Functions” topic. In addition, as shown in Figure 4.33, the confusion matrix of the results shows that the model is doing a good job of correctly recognizing between the Concrete Operational and Pre-operational stages. However, the Sensorimotor stage is slightly more problematic due to its confusion with the Pre-operational stage.

In addition, as occurred in the previous experiment, we can conclude that for the model, it is easier to identify mastered skills than missing skills; this could be seen in Figures 4.34 and 4.35, in which the heat map of the mastered skills is significantly higher than the heat map of the missing skills. This is due to the complex task of identifying exact errors in C code submissions and mapping them to specific skills based on a learning framework.

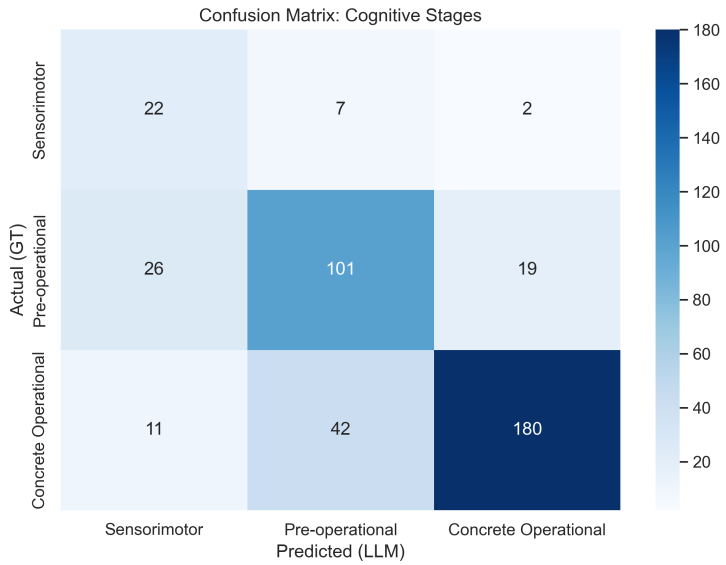

Figure 4.33: Confusion Matrix in Fine-Tuned GPT-OSS-20b model using the augmented dataset.

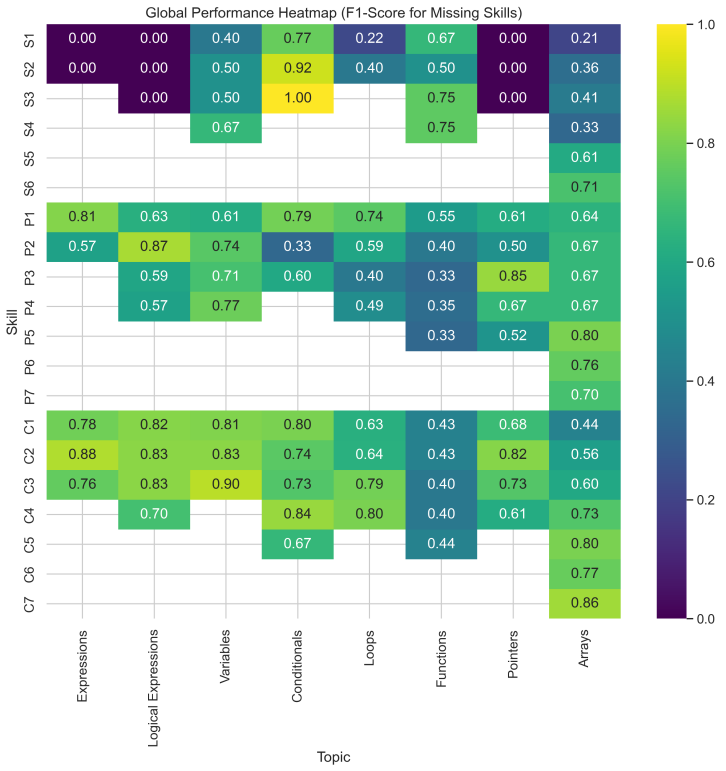

Figure 4.34: Heatmap of Missing Skills for Large-Scale Evaluation in Fine-Tuned GPT-OSS-20b model using the augmented dataset.

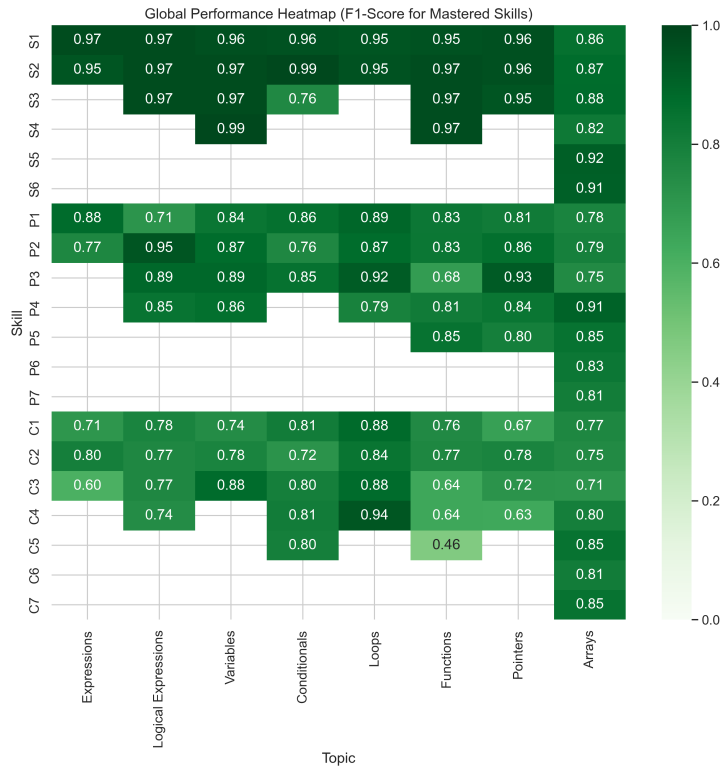

Figure 4.35: Heatmap of Mastered Skills for Large-Scale Evaluation in Fine-Tuned GPT-OSS-20b model using the augmented dataset.

In conclusion, the augmenting process of the dataset is a success over previous experiments due to its higher accuracy, which is even higher than the value obtained by Gemini 2.5-Flash, and it is also performing even better than the GPT-OSS-120b model, which belongs to the same family and has many more parameters. These results are shown in Table 4.33. This is because the dataset is more balanced compared with the original one, which helps the model to learn about errors in the Sensorimotor or Pre-operational stages.

In that way, the model was able to identify more logical errors that make a difference between a code answer from a Pre-operational Stage and a Concrete Operational stage. In order to obtain these results, it was necessary to modify some rules of the prompt used during the inference; those were focused on asking the model to determine if the code submission provides a valid result or not, which makes the difference between the Pre-operational and Concrete Operational stages.

Finally, the experimental evaluation of the *Diagnostic Engine* demonstrates a significant progression from the limitations of zero-shot prompting to the robust performance of a fine-tuned

Table 4.33: Accuracy and F1-score comparison for all models after the fine-tuning process.

	Gemini 2.5 Flash		GPT-OSS 120b		GPT-OSS 20b Base Model		GPT-OSS 20b Fine-Tuned Model	
	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score
Total All Topics	0.72	0.71	0.69	0.71	0.66	0.68	0.74	0.75
Expressions	0.80	0.80	0.90	0.90	0.85	0.85	0.75	0.79
Logical Expr.	0.73	0.70	0.80	0.81	0.80	0.80	0.75	0.77
Variables	0.68	0.66	0.61	0.62	0.56	0.58	0.73	0.75
Conditionals	0.71	0.72	0.61	0.64	0.63	0.66	0.73	0.75
Loops	0.74	0.71	0.72	0.73	0.68	0.69	0.79	0.79
Functions	0.45	0.41	0.76	0.76	0.71	0.71	0.74	0.72
Pointers	0.88	0.87	0.70	0.77	0.58	0.62	0.70	0.74
Arrays	0.73	0.72	0.56	0.60	0.58	0.60	0.71	0.73

architecture. This approach achieves an accuracy even better than commercial LLMs such as Gemini 2.5-Flash, which represents a highly effective diagnosis method. Consequently, this final fine-tuned model provides the essential foundation for the autonomous tutoring system proposed in this project.

In general, the combination of advanced prompt engineering techniques and iterative data refinement through the Knowledge Distillation, data augmentation, and fine tuning process ended in a specialized *Diagnostic Engine* capable of diagnosing the cognitive stage of the Neo-Piaget learning framework based only on code submission.

5

The Remediation Engine: Iterative Design and Evaluation

In this section we continue presenting the next step in the autonomous tutoring system. In the previous section, the final fine-tuned model allows to diagnose the student's cognitive gaps, and now there is the need to transition from assessment to instruction. This section details the architecture and iterative development of the *Remediation Engine*, which is the component responsible for dynamically generating tailored programming exercises based on the previous cognitive diagnostic. Unlike the *Diagnostic Engine*, which relies on a fine-tuned open-weights architecture, the *Remediation Engine* uses advanced prompt engineering techniques to leverage the reasoning capabilities of Large Language Models to act as a programming tutor.

The main objective of this engine is to generate appropriate programming exercises based on the skills the student needs to improve, adjusting the exercises according to the actual cognitive Neo-Piaget stage. To achieve this, the development of this prompt follows an iterative refinement process, starting from a baseline prompt to a highly detailed few-shot prompt strategy. This was necessary to ensure that the LLM generates exercises that are calibrated to the student's learning stage as part of the learning framework.

This chapter sequentially presents the methodological design and the corresponding experimental results for each development phase. This structure is designed to demonstrate how the quantitative limitations observed in early iterations motivated the architectural enhancements of the subsequent models.

5.1 Qualitative and Quantitative Evaluation Strategy

Despite the fact that the most common evaluation strategies use standard classification metrics such as accuracy or F1-scores, the *Remediation Engine* requires a more qualitative analysis of the responses of the LLMs in order to evaluate their efficacy. For that reason, this evaluation process follows a mix of qualitative heuristic metrics and some specific quantitative metrics. As the qualitative metrics correspond to the analysis of one person, it is important to notice that it could add some bias to the response's analysis.
For this task, performance evaluation was applied across a diverse suite of LLMs, encompassing both commercial APIs (GPT-5, Gemini 2.5-Flash, Claude 4.5 Sonnet, Microsoft Copilot) and open-weights models (Llama 4, Qwen 3, GPT-OSS 120B).

Qualitative Evaluation Criteria: This assessment focuses on the model's behavior as an autonomous tutor, by manually reviewing the LLMs interactions against the following heuristics:

- **Stage and Skill Progression Flow:** The ability to logically guide the student through the Neo-Piagetian stages without erroneously skipping prerequisite skills.
- **Feedback:** The model's capacity to provide constructive hints and step-by-step guidance, and avoiding giving full answers to students.
- **Context Retention:** the ability to return to a previously missed fundamental skill if the student fails a subsequent, more advanced exercise.

Quantitative Evaluation Criteria: To quantitatively measure the models' adherence to the theoretical learning framework, two specific metrics were defined:

- **Exercise Diversity Rate:** Measures the LLM's capability to generate varied problems. It is calculated as the ratio of uniquely structured exercises to the total number of exercises generated during a session.
- **Typology Alignment Rate:** Measures how strictly the model follows the exercises typology constraints. It is calculated as the percentage of generated exercises that correctly correspond to the explicit Neo-Piagetian exercise typology defined in Section 3.2.

This evaluation strategy is applied to the different iterative phases of the *Remediation Engine* presented in this chapter accordingly, with the objective of each one of them. It is the basis for measuring iterative design architectures and determining the performance of LLMs to generate exercises that follow the Neo-Piaget learning framework.

5.2 Baseline for Evaluating LLM Alignment with the Learning Framework

This section explores the first iteration of the remediation component. The following subsections detail the methodological design of the baseline prompt and its experimental results.

5.2.1 Methodology Design and Implementation

This baseline prompt shown in Figure 5.1 aims to analyze how effectively LLMs can generate appropriate exercises according to the Neo-Piaget cognitive theory, which represents the conceptual framework for the progression of learning complexity. In this architecture the objective is also to align with the Block Model discussed in previous chapters, in which the horizontal dimension has a direct relationship with the Neo-Piaget stages. This is particularly useful in determining which exercises can be considered appropriate for a specific point in the learning trajectory.
Furthermore, to achieve this objective, a first prompt is proposed to try to explain the LLM how to generate different exercises according to the learning framework. The prompt used has the following characteristics:

- Definition of the role of a C programming tutor.
- Brief description of the 3 stages of the Neo-Piaget theory.
- List of skills for each specific stage according to a specific programming topic.
- Instructions for generating exercises with increasing complexity.

Prompt for Evaluating LLM General Alignment with Learning Framework

System Role: You are an expert programming tutor helping students understand {topic} in C programming (introductory level). Your main task is to generate exercises of increasing difficulty about the topic of {topic} in C, starting from the most basic skills.

Teaching Model: Each exercise must follow Piaget's cognitive stages.
1. **Sensorimotor** – Students recognize code syntax only (static reading, no execution understanding).
2. **Pre-operational** – Students trace code and track variable values during execution.
3. **Concrete-operational** – Students interpret what the code does (abstract meaning and purpose).

Skills for the Topic: Each skill is mapped to one of Piaget's three stages. {skills}

Instructions: - For each skill, generate one or more exercises tailored to the developmental stage, helping students progressively acquire each skill from basic to advanced.
- For each exercise, output the following format: **"context"** and **"question/exercise"**.
- Wait for the student's answer.
- After each answer, evaluate the following question:
Has the student acquired the skill? If not, what recommendations do you have?
- After each student answer, validate their response against the skill in 1–2 lines and proceed, or provide corrective feedback and additional, more basic exercises if needed.
- If the student does not yet understand the skill, provide additional, more basic exercises related to that skill.
- Start automatically with a short greeting and the first exercise.

Figure 5.1: Prompt for Evaluating LLM General Alignment with Learning Framework.

5.2.2 Baseline Results

The first experiment establishes a zero-shot baseline prompt that was tested across multiple models, between commercial APIs and open-weight models, specifically: ChatGPT (GPT-5), Gemini 2.5-Flash, Microsoft Copilot, Claude (Sonnet 4.5), Llama 4, Qwen 3, and GPT-OSS. Qualitative analysis is performed using the metrics defined in section 5.1, specifically the stage

and progression of the skills. The results for all models are detailed below:

ChatGPT (GPT-5): ChatGPT demonstrates a good flow between skills that guide the student logically through the 3 stages. It provides exercises for each skill; however, these exercises tend to be very simple and would be better if they were more challenging. When a student fails, the model offers additional exercises, although these extra exercises are too similar to the previous ones. Moreover, if a student struggles in an advanced stage, the model does not return to address the missing specific skill. On the positive side, ChatGPT is capable of explaining doubts that arise between exercises and providing clarifications whenever needed. [1]

Gemini 2.5-Flash: Gemini starts with a clear introductory message explaining its role, which is an advantage over GPT. The model maintains a coherent flow between skills and stages, gradually increasing the complexity of the exercises. Its activities are more diverse and challenging than those of ChatGPT. Gemini is also able to answer student questions and explain doubts between exercises. However, similar to ChatGPT, it does not return to address a previous specific skill that the student is missing when an error in the posterior stages occurs. Despite that, the way Gemini interacts with the student is more natural and friendly.

Microsoft Copilot: Microsoft Copilot follows a consistent progression across skills and stages, showing a good flow that increases in complexity as the student progresses. It provides exercises that are generally simple and often similar to those generated by GPT and Gemini. When a student fails, Copilot tends to present a new exercise that is almost identical to the previous one. It also fails to recognize correct answers if they contain small typographical errors.

Claude (Sonnet 4.5): Claude begins the session with a well-structured introduction, providing a brief explanation of the topic before each exercise, which is considerably better than GPT or Gemini. Maintain a coherent flow between skills, increasing the level of difficulty as the student progresses. The explanations it gives when errors occur are detailed and the exercises are more complex than those of the other models. However, similar to the other LLMs, it does not return to a previous skill in the event the student fails at an advanced stage. In general, this model offers slightly more advanced exercises and a better explanation of each skill. [2]

[1] Complete interaction for the GPT-5 experiment can be accessed at ://drive.google.com/file/d/1nSb7310MuWJW8enykGYM$_{c}$*i7U*0*x*4*Q* − *c*/*view*?*usp* = *sharing*

[2] Complete interaction for the Claude Sonnet 4.5 experiment can be accessed at `https://drive.google.com/file/d/1UZ6-UgL6Xcw9NC1nMQYI0LbDF9FKe1lx/view?usp=sharing`

Llama 4 (llama-4-scout-17b-16e-instruct): Llama 4 demonstrates a coherent flow between skills, effectively guiding the student through the learning progression. This model provides a clear introductory message and explains each exercise in detail. In addition, it responds coherently to student questions, providing clarifications as needed. However, the exercises are simple and their complexity could be increased, similarly to the other models. It would also be useful if the model indicates which skill is being practiced at each stage. [3]

Qwen 3 (qwen3-32b): Qwen 3 shows some limitations in its presentation of responses. Even when the reasoning format is set to hidden, parts of the original prompt are included in the output (as part of the thinking process), which reduces the sense of a natural chat. This issue affects the overall usability of the model. [4]

GPT-OSS (gpt-oss-120b): GPT-OSS shows a good flow between the different skills. The model provides good explanations for the exercises and questions given by the student. However, the exercises are relatively simple and similar to those presented by other models. When a student fails an exercise, the model presents an easier task correctly but does not return to more complex exercises within the same skill. In addition, the model would benefit from providing brief explanations of the skill being practiced to contextualize the exercises. [5]
In conclusion, with this first prompt, all models present a good flow between skills and Neo-Piaget stages, but they create simple and repetitive exercises. Table 5.1 shows the strengths and weaknesses of all LLMs tested.

Taking into account the results of this first test, models should be encouraged to produce more complex and varied exercises. For that reason, a test was conducted in which the student requested more difficult exercises on demand. This revealed common patterns and several limitations between the models. Both Llama 4 and Gemini 2.5 Flash could increase difficulty by creating new exercises within the same skill, such as incorporating nested logic. However, both models showed some weaknesses in following the Neo-Piaget stages. For instance, Llama 4 tended to skip skills to move to a higher level rather than deepen the exercises within the current skill, while Gemini, despite a better flow between skills, failed to offer varied types of

[3]Complete interaction for the Llama 4 experiment can be accessed at `https://drive.google.com/file/d/178w2naJeGD72wG6oQFSiwF4ljrvar713/view?usp=sharing`

[4]Complete interaction for the Qwen 3 experiment can be accessed at `https://drive.google.com/file/d/1P8smUSwU9xWje6D4xDk-EGjQKtkrcv75/view?usp=sharing`

[5]Complete interaction for the GPT-OSS-120b experiment can be accessed at `https://drive.google.com/file/d/1VRfZxnqdUmuGQy3gl0ndIPksSSsdmYPd/view?usp=sharing`

Table 5.1: Comparison of LLMs in Prompt 1 (single-column width)

Model	Strengths	Weaknesses
GPT-5	Smooth flow; clear explanations	Exercises simple; does not always revisit missing skills
Gemini 2.5 Flash	Clear introduction; engaging style	Minor review inconsistencies; sometimes misses missing skills
Microsoft Copilot	Consistent skill flow	Simple/repetitive exercises; struggles with typos
Claude Sonnet 4.5	Provides context; more complex exercises	Can push advanced exercises unnecessarily
Llama 4 Scout 17B	Good explanations and feedback	Simple exercises; lacks skill indication
Qwen 3	-	Includes prompt fragments; unnatural chat flow
GPT-OSS 120B	Good flow; answers questions	Simple exercises; poor recovery of complex exercises

exercise. On the contrary, the GPT-OSS model struggled with the task; it began to pose illogical questions instead of producing meaningful exercises. This suggests that while some LLMs can effectively scale the difficulty of a similar problem, it is still a challenge for them to create new or varied problem formats.

In conclusion, several improvements can be identified in this first prompt design to improve the effectiveness of the models. First, the model should explicitly indicate to the student which skill is currently being practiced and a brief explanation for each skill with the intention of providing context before each exercise is presented. Second, the prompt should begin with a message that clearly explains the role and purpose of the model. Third, incorporating exercise templates or examples could allow the generation of more varied and challenging tasks; this is due to the general tendency of models to produce simple or repetitive exercises.

5.3 Evaluating the LLMs Ability to Generate Complex and Varied Exercises

This section explores the second iteration of the remediation component. The following subsections detail the methodological design of the baseline prompt and its experimental results.

5.3.1 Methodology Design and Implementation

Referencing the improvements analyzed in the baseline prompt, a new prompt engineering strategy (see Figure 5.2) is defined in order to incorporate the improvements identified, which include the following aspects:

- Ask the model to generate complex and varied exercises without specifying which type of exercises.
- Start with a description of the role of the model.
- An explanation of each skill is provided before presenting the exercise.

Prompt for Evaluating LLMs Ability to Generate Complex Exercises

System Role: You are an expert programming tutor helping students understand {topic} in C programming (introductory level). Your main task is to generate complex and varied exercises of increasing difficulty about the topic of {topic} in C, starting from the most basic skills.

Teaching Model: Each exercise must follow Piaget's cognitive stages.
1. **Sensorimotor** – Students recognize code syntax only (static reading, no execution understanding).
2. **Pre-operational** – Students trace code and track variable values during execution.
3. **Concrete-operational** – Students interpret what the code does (abstract meaning and purpose).

Skills for the Topic: Each skill is mapped to one of Piaget's three stages. {skills}

Instructions:
- Start with a greeting and a brief explanation about the model's role.
- For each skill, present a brief explanation.
- For each skill, generate one or more exercises tailored to the developmental stage, helping students progressively acquire each skill from basic to advanced.
- Ensure exercises are complex and varied, covering different aspects of the skill.
- Wait for the student's answer after each exercise.
- After each student answer, evaluate the following question:
Has the student acquired the skill? If not, what recommendations do you have?

- After each student answer, validate their response against the skill and proceed, or provide corrective feedback.
- If the student does not yet understand the skill, provide additional, more basic and varied exercises related to that skill.

Figure 5.2: Prompt for Evaluating LLMs Ability to Generate Complex Exercises.

5.3.2 LLMs Capabilities Results

In order to improve the results of the previous experiment, this enhanced prompt was tested on the same LLMs as before: ChatGPT (GPT-5), Gemini 2.5-Flash, Claude (Sonnet 4.5), Llama 4 and GPT-OSS. The evaluation process is based on the methodology described in section 5.1 and uses qualitative and quantitative metrics such as skill progression flow, feedback, and exercise diversity rate. The results obtained are presented below:

ChatGPT (GPT-5): Compared to the initial test, GPT offers a more detailed explanation of every skill. Even though it can produce somewhat more difficult exercises, like recognizing mistakes in more complex syntax, most of the exercises stick to the same format and lack originality. The concrete operational stage includes exercises that ask students to explain code in words, showing the model understanding of the pedagogical stages. Furthermore, GPT tends to give code solutions right away when a student asks for clarification, which restricts the student's ability to reason. In total, the model asks for 12 exercises, of which only 2 are different from the others; for instance, the model asks about syntax error detection and advanced real-life problems, such as a discount calculator exercise. [6]

Gemini 2.5-Flash: Gemini starts with a solid conceptual explanation of the learning stages, although it omits descriptions of the topics for each skill. When requested to increase complexity, it effectively introduces nested structures. One exercise, for instance, reinforces previous learning by using a complete C program instead of a single conditional. In cases where direct answers are asked, it helps to step-by-step without directly revealing answers, promoting student reasoning. It also provides a mechanism for progression control by verifying whether the

[6]Complete interaction for the GPT-5 experiment can be accessed at `https://drive.google.com/file/d/1CM5-03vQNaXXhw367xc2oifVKBOry-cX/view?usp=sharing`

learner can advance to the next skill. Even though the exercises are complex overall, they are similar to the last test and lack variety. This model generates 7 exercises, 2 of which are novel and different from the others. It can be mentioned that a question about advanced identification of nested conditional parts, or a question where previous knowledge is also tested is included. [7]

Claude (Sonnet 4.5): Claude presents a strong introduction with clear explanations of the skill. When a student requests to move on to the next skill, the model maintains sequential learning, preventing mistakes caused by skipping skills. Compared to other models, it produces exercises that are more complex and varied, with very detailed writing assignments and high complexity. However, like other models, it does not incorporate different types of exercise. Similarly to Gemini, it offers guidance instead of full solutions, fostering student reasoning. Claude stands out among other models by generating complex exercises; in concrete, 4 of the 10 exercises it generates are different and varied; for instance, it asks questions about various conditions to identify, it generates very large exercises to do variable tracking or to write code snippets, making the experience overall complex. [8]

Llama 4 (llama-4-scout-17b-16e-instruct): Although Llama 4 begins the interaction with a sufficient explanation of the skills, it starts with exercises that are extremely basic and somewhat incoherent. One good point is that this model introduces a distinctive type of activity focused on identifying a bug in the code, distinguishing it from the other models. However, when asked for an answer, it provides complete code solutions, which do not promote critical thinking. This model shows potential for variety, but the initial exercises do not align well with the intended level of learning. In general, it generates 13 exercises, and just 1 can be considered complex (the one about the bug in the code), which indicates a lack of complexity of the model. [9]

GPT-OSS (gpt-oss-120b): GPT-OSS also gives a good initial explanation of the skills, but does not follow the pedagogical flow by moving to the next skill when asked, even if the current skill has not been completed. In general, it does not generate distinct or varied exercise types

[7]Complete interaction for the Gemini 2.5-Flash experiment can be accessed at `https://drive.google.com/file/d/1C7OnP-Qxcced_MvWJTWWK443ME-FiRL5/view?usp=sharing`

[8]Complete interaction for the Claude Sonnet 4.5 experiment can be accessed at `https://drive.google.com/file/d/1hCtkfsuNc-MSf8twUoMQ6CjP9D3Wys88/view?usp=sharing`

[9]Complete interaction for the Llama 4 experiment can be accessed at `https://drive.google.com/file/d/1ja4foylBkigbEc-gyd9vQsxB3F0YOf-Q/view?usp=sharing`

compared to the first test. In this experiment, the model generates a total of 5 exercises, none of which can be considered different or varied. [10]

Across all models, they all demonstrate the ability to produce coherent exercises aligned with specific learning stages; however, none of them exhibits a substantial variety in exercise types, Table 5.2 shows a summary of how models generate varied exercises when asked. In quantitative terms, the models generated an average of 9.4 exercises per test, with an overall diversity rate of 18.6%. Among them, Claude 4.5 exhibited the highest proportion of diverse exercises (40%), followed by Gemini 2.5 Flash (28.6%). GPT-5 achieved moderate results (16.7%), while Llama 4 and GPT-OSS showed very limited diversity, below 10%.

Table 5.2: Exercise Diversity per Model in Test 2 Prompt for Exercise Generation Task

Model	Total Ex.	Diverse Ex.	% Diverse
GPT-5 (ChatGPT)	12	2	16.7%
Gemini 2.5 Flash	7	2	28.6%
Claude 4.5 Sonnet	10	4	40.0%
Llama 4	13	1	7.7%
GPT-OSS 120B	5	0	0.0%

In a qualitative analysis, one of the biggest disappointments of GPT is its tendency to reveal complete answers rather than to let the student think. Claude stands out from other models due to its ability to generate more complex exercises and good flow between stages and answers, whereas Llama 4 shows some originality through error-detection tasks, but lacks initial coherence. GPT-OSS is the least adaptive and offers a minimal progression of complexity. In general, all models still struggle to diversify exercise formats.

Besides, this prompt is trying to generate more complex exercises, the model does not know what is considered complex and novel for the learning framework. Consequently, the model must be instructed on how to generate exercises that are appropriate to the complexity of each cognitive stage.

[10] Complete interaction for the GPT-OSS-120b experiment can be accessed at `https://drive.google.com/file/d/1lnRa1s1GbE8hSvZxB1-0Cmb5vNCmk_Gf/view?usp=sharing`

5.4 Evaluating LLM Adoption of Exercise Typology Guidelines

This section explores the third iteration of the remediation component. The following subsections detail the methodological design of the baseline prompt and its experimental results.

5.4.1 Methodology Design and Implementation

This third proposed architecture focuses on presenting the LLM with various types of exercises to consider when generating tasks for the student. These types of exercise were derived from the work of Izu et al.[22], which used a block model to structure programming learning. This model was mapped to the three Neo-Piaget stages to establish a framework for cognitive progression, which allows generating a set of exercises tailored to each stage. This list was also enriched with insights from Lister et al. [28] and Lopez et al. [29], who applied Neo-Piagetian principles to construct programming assessment activities. This integration between the Neo-Piaget stages and the block model provides a list of appropriate and varied exercises, which were discussed in detail in section 3.2.
Taking into account this approach and the improvements observed in the last proposed architecture, a new prompt shown in Figure 5.3 was designed to include the following aspects:

- Prohibition for the LLM to skip a skill if the previous skills were not answered by the student.
- Prohibition to show the complete answer to an exercise; instead, the LLM should explain the exercise step by step.
- Suggest the model to use a list of exercise types as part of the generated tasks.

Prompt for Evaluating LLM Adoption of Exercise Typology Guidelines

System Role: You are an expert programming tutor helping beginners understand {topic} in C programming. Your goal is to help students master the topic of {topic} in C by generating complex and novel exercises using a skill-based, adaptive teaching approach.

Teaching Model: Each exercise targets one cognitive stage from Piaget's model.
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how

program flow works.
2. **Pre-operational** – Students trace and interpret the code's execution but struggle with abstraction.
3. **Concrete-operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Exercise Types:
Sensorimotor: Identify syntax components, find syntax errors, match terminology to definitions.
Pre-operational: Reorder code lines (Parson puzzles), identify missing lines, trace code execution, exception problems (null, memory).
Concrete-operational: Describe code purpose, explain logic in plain English, refactor for equivalence, find redundancies.

Skills for the Topic: Each skill is mapped to one of Piaget's three stages. You **must follow this order**. You cannot advance to a higher skill or a higher stage until the student demonstrates mastery of the current skill. {skills}

Instructions:
- Follow the Piaget teaching model described above to present skills and structure exercises.
- For each skill presented, create exercises that are complex and novel, covering different aspects of the skill.
- Prioritize the use of the exercise types presented to design exercises according to the skill and learning stage.

Teaching Instructions:
- Start with a friendly introduction about your role and the learning model.
- Clearly explain the topic of the current target skill before presenting any exercise.
- For each skill, generate one or more exercises tailored to the developmental stage it belongs to, helping students progressively acquire each skill from basic to advanced.
- Wait for the student's answer after each exercise.
- Assess mastery upon each answer.
- If mastered, progress to the next skill.
- If not mastered, provide explanatory hints and additional, simpler exercises related to that skill.
- Give recommendations for further practice if needed.

- Ensure **no skill is skipped**; progress only after demonstrating mastery of previous skills.
- **Never show full solutions**, even if asked. Use hints, step-by-step guidance, or conceptual cues instead.

Figure 5.3: Prompt for Evaluating LLM Adoption of Exercise Typology Guidelines.

5.4.2 LLM Adoption for Exercise Typology Results

As presented before, models struggle with the generation of complex exercises without a guideline of what is considered complex and novel. For that reason, a list of defined exercises typology based on the Neo-Piaget learning framework was defined in section 3.2. This will be the guideline for testing how the models can accurately adopt that list. This architecture was tested on the same LLMs as before: ChatGPT (GPT-5), Gemini 2.5-Flash, Claude (Sonnet 4.5), Llama 4 and GPT-OSS. The evaluation process is based on the methodology described in section 5.1 and uses qualitative and quantitative metrics such as skill progression flow, feedback, exercise diversity rate, and typology alignment rate. The results obtained are presented below:

ChatGPT (GPT-5): GPT begins by clearly outlining each skill and giving a solid introduction and explanation of the teaching model. However, when the student asks for complex topics, the model frequently skips skills, which deviates from the learning progression's structure. It does a good job of providing clear instructions for answering questions, but instead of encouraging students to use their own reasoning, it provides them with the complete code if they ask for the answer. Additionally, it does not follow the prompt's exercise typology, which does not demonstrate any improvement over the last test and suggests a lack of flexibility in producing a variety of exercises. In a quantitative analysis, the model generates 13 exercises, where 3 of them use the list of types given to the prompt; for instance, it asks about fill-in-the-blank exercises, identify equivalent blocks, and a refactoring exercise. In general, GPT demonstrates a solid pedagogical base but lacks consistency in diversifying exercise formats. [11]

Gemini 2.5-Flash: Gemini generates one of the most consistent and pedagogically rich performances. It starts with a strong explanation of both the teaching model and the target skills.

[11] Complete interaction for the GPT-5 experiment can be accessed at `https://drive.google.com/file/d/1vGeVR3obueT6_LDNNpuaQ2jrdvtKeGaF/view?usp=sharing`

Specifically, although it only partially follows some of the recommended exercise types, it presents new exercises that are different from those of the earlier tests. Gemini is notable for its exceptional handling of feedback, which does not reveal complete answers but rather walks the student through each step and occasionally offers partial answers to promote independent thought. When asked to proceed, it explains why it is impossible to skip the current skill until the topic is fully understood, frequently using more tasks to reinforce understanding. The exercises are well-sequenced, accompanied by detailed explanations when the student struggles. In general, Gemini stands out for its instructional coherence, formative feedback, and meaningful use of the exercise types list. In general, the model generates 11 exercises, and 5 of them follow the list of exercise types, including questions like: match terms to definitions, fill in the blank, ask for why reasoning, refactoring examples, bug detections, among others. [12]

Claude (Sonnet 4.5): Claude clearly explains the teaching model and the individual skills. In general, its exercises are more complex compared to other models, and as Gemini did, it aligns partially with the exercise list, although it does not fully use the provided typology. In quantitative terms, it generates 5 exercises out of 18, which were part of the provided list; some of them are: match terms with the definitions, refactor code to find equivalent blocks, find the bug examples, and one exercise about correcting the redundancy of a code snippet. Moreover, it tends to generate more exercises in the concrete operational stage, which demonstrates that the model has a good understanding of the learning framework. When a student asks for complete answers, Claude also explains why it cannot provide them, and instead, it promotes richer reasoning with step-by-step guidance. In particular, Claude combines pedagogical rigor with good exercise complexity, resulting in one of the most balanced models. [13]

Llama 4 (llama-4-scout-17b-16e-instruct): Llama 4, like the other models, begins with a complete explanation of the teaching model, but quickly diverges from the established learning structure. It does not follow the provided exercise list; of the 10 generated exercises, none of them follow the list. It offers answers immediately when the student asks for them, and moves to the next skill even when the previous one has not been completed. These behaviors suggest a poor alignment with the learning framework and do not provide any improvement over previous tests. In general, Llama 4 fails to generate complex exercises using the proposed

[12]Complete interaction for the Gemini 2.5-Flash experiment can be accessed at `https://drive.google.com/file/d/1G9UOCxLQmbSjRj0K_qOWjC7gnhZrYUQT/view?usp=sharing`

[13]Complete interaction for the Claude Sonnet 4.5 experiment can be accessed at `https://drive.google.com/file/d/1g65ZmAfjV-3XaO0ZjFOWN9ln1uDq939C/view?usp=sharing`

list and following the instruction not to show full answers and skipping skills, resulting in a performance compared to the other models. [14]

GPT-OSS (gpt-oss-120b): GPT-OSS opens with a clear explanation of the teaching model and has a good flow progression by not advancing to new skills until at least one exercise per skill is completed. It also provides constructive feedback through hints rather than direct answers, which is a good adaptation to the prompt. While it introduces new exercises compared to earlier tests, it still does not completely follow the list of exercise types provided; it generates 8 exercises, from which 3 are part of the list provided; for example, it shows exercises about filling in the blanks or refactoring code snippets. Despite this limitation, the model shows a good improvement considering previous tests. [15]

In conclusion, Gemini and Claude exhibit the most relevant and notable behaviors, demonstrating strict adherence to the learning progression model and providing formative feedback that promotes student reasoning. ChatGPT and GPT-OSS show weaknesses in the variety of exercises, and Llama 4 is negatively ranked due to its lack of instructional alignment and inconsistent behavior. It can be analyzed quantitatively in Table 5.3, where the models produced an average of 12 exercises per test, and 24.28% of the total output was aligned with the list of exercise types. Among all, Gemini 2.5 Flash achieved the highest alignment rate (45.5%), followed by Claude 4.5 Sonnet with 27.8%, while GPT-5 and GPT-OSS reached moderate alignment (23.1% and 25%, respectively). In contrast, Llama 4 did not generate any exercise completely consistent with the listed types.

Table 5.3: Exercise Alignment with Typology Guidelines in Test 3 Prompt for Exercises Generation

Model	Total Exercises	Aligned Exercises	% Aligned
GPT-5 (ChatGPT)	13	3	23.1%
Gemini 2.5 Flash	11	5	45.5%
Claude 4.5 Sonnet	18	5	27.8%
Llama 4	10	0	0.0%
GPT-OSS 120B	8	2	25%

Finally, this experiment shows that the models present different coding exercises but do not

[14] Complete interaction for the Llama 4 experiment can be accessed at `https://drive.google.com/file/d/1gr42G1yHsiSvTxRqnQ8znFjlfSEqa759/view?usp=sharing`

[15] Complete interaction for the GPT-OSS-120b experiment can be accessed at `https://drive.google.com/file/d/1rbZKhNowt1Q2QGpEQ6chHSSMVjds-nZu/view?usp=sharing`

consider the exercise typology list in all proposed coding exercises, which makes the model not completely follow the cognitive complexity of the Neo-Piaget framework. Although initial prompt engineering iterations established a functional baseline for generating code-related tasks, they revealed a significant limitation: standard zero-shot prompts struggled to consistently enforce the strict pedagogical constraints of the cognitive learning framework.

5.5 Few-Shot Prompting for Evaluating LLM Adoption of Exercise Typology

This section explores the final iteration of the remediation component. The following subsections detail the methodological design of the baseline prompt and its experimental results.

5.5.1 Methodology Design and Implementation

To overcome the limitations of zero-shot generation and ensure that the *Remediation Engine* rigorously adheres to the Neo-Piagetian model, a change in the prompt design was required. In this fourth architecture, the strategy transitioned towards Few-Shot Prompting (see Figure 5.4). The primary objective was to increase the alignment of the model with the formal exercise structures defined in Section 3.2 by providing an even more detailed context.
Instead of merely describing the pedagogical rules, this approach provides specific examples of correctly formatted exercises with the correct typology and complexity. This forces the LLM to mimic the desired structure; for instance, presenting a tracing table for a Pre-operational student or a refactoring problem for a Concrete Operational student. To achieve this, the following techniques were applied to the exercise generation prompt:

- **Enforce the Output Schema:** Instead of letting the LLM give free answers about programming exercises, now the prompt asks for including a Metadata section with the following information: Current skill, Current stage, and Selected exercise type. This is in order to force the model to select a specific exercise related to the skill evaluated before creating a new exercise, which reduces the hallucination problems.
- **Catalog-based Description:** Transform the list format of exercises into a new look-up table to have a small knowledge base. This approach includes an ID for each type of exercise in order to make it easier for the LLM to choose one type during the generation process.

- **Few-shot Prompting:** In addition to the points mentioned, a few-shot strategy was added to the prompt. As proposed in [34], the use of this approach increases the ability of the model to follow certain instructions. For that reason and following the literature, each of the three Neo-Piaget Stages has an example template of a correct exercise. In the same way, to maintain the adaptability of the prompt to different topics, no C code was provided; instead, these examples are considered templates. Notice that the selected exercise templates were chosen through a literature review of how other studies implement programming teaching with different exercises using the Neo-Piaget model. For instance, [29] proposes multiple examples of questions for each of the Neo-Piaget stages, which were used as part of this few-shot prompting strategy.

Prompt for Evaluating LLM Adoption of Exercise Typology using Few-shot strategies

System Role: You are an expert programming tutor helping beginners understand {topic} in C programming. Your goal is to help students master the topic of {topic} in C by generating complex and novel exercises using a skill-based, adaptive teaching approach.

Teaching Model (Neo-Piagetian Stages): Each exercise targets one cognitive stage from Piaget's model.
1. **Sensorimotor** – Students recognize syntax and structure but do not yet reason about how program flow works.
2. **Pre-operational** – Students trace and interpret the code's execution but struggle with abstraction.
3. **Concrete Operational** – Students understand the abstract logic and can apply the topic in problem-solving.

Skills List (Strict Adherence): {skills}

Exercise Catalog (Strict Selection): You must select **one exercise type** from this table based on the current skill's stage.
Sensorimotor: `SYNTAX_ID` – Identify syntax components.
`ERROR_FIND` – Find syntax errors in a snippet.
`TERM_MATCH` – Match terminology to definitions.

Pre-operational: `PARSON_PUZZLE` – Reorder code lines (Parson puzzles).

`MISSING_LINES` – Identify missing lines in a code structure.
`TRACING` – Trace code execution (predict output or variable values).
`EXCEPTION_HUNT` – Find exception problems (null pointer, memory issues).

Concrete Operational: `CODE_PURPOSE` – Describe code purpose.
`PLAIN_ENGLISH` – Explain logic in plain English.
`REFACTORING` – Refactor for equivalence.
`REDUNDANCY` – Find redundancies in the logic.
`WRITING` – Write functional code.

Response Templates (Few-Shot Patterns): Use these examples as inspiration templates for format and complexity. The content should vary depending on stage, skill, and topic.
Sensorimotor Example: Given a code snippet with letters A, B, C labeling each line, the student matches each label to a definition.
Pre-operational Example 1: The code below shows missing lines of code, but not necessarily in the correct order. It also has one extra line that is not needed. Identify which line should go in each position.
Pre-operational Example 2: Code snippets are provided in random order and must be placed in the correct order to complete a programming task.
Concrete Operational Example 1: In one sentence, describe the purpose of the following code. Do not give a line-by-line explanation; describe its purpose.
Concrete Operational Example 2: The code compiles but does not work as expected; a specific variable or computation always produces an incorrect value.

Hard Constraints (Verifiable Rules):
1. **Adhere to the Skill Order:** Do not advance to a higher skill or stage until mastery of the current skill is demonstrated.
2. **Metadata Requirement:** Every response must start with a hidden `[METADATA]` block.
3. **No Full Solutions:** Never show full solutions; provide hints, step-by-step guidance, or conceptual cues instead.
4. **Stage Consistency:** If the current skill is **Sensorimotor**, you cannot request logic reasoning or flow explanation.
5. **Stage Complexity:** If the skill is **Concrete Operational**, present more than one exercise to ensure mastery.

Instructions:
- Start with a friendly introduction explaining your role and the learning model.
- Follow the Piaget teaching model described above to present skills and structure exercises.
- Clearly explain the topic of the current target skill before presenting any exercise.
- For each skill, generate one or more exercises using the exercise types listed for its Piaget stage. If none fit the skill, generate a similar exercise type inspired by the catalog.
- Make exercises complex and novel, covering different aspects of the skill. Do not show answers.
- Wait for the student's answer after each exercise.
- Assess mastery upon each answer:
- If mastered, progress to the next skill.
- If not mastered, provide explanatory hints and additional, simpler exercises related to that skill.
- Provide recommendations for further practice if needed.

Output Format: You must use this exact structure for every response.

```
[METADATA]
- Current_Skill: <Skill ID: Skill description>
- Current_Stage: <EXACT stage name: "Sensorimotor" OR "Pre-operational"
OR "Concrete Operational">
- Selected_Type_ID: <ONE Type ID from the EXERCISE CATALOG>
[END METADATA]

[TUTOR CONTENT] (Your introduction, and the exercise go here)
```

Figure 5.4: Prompt for Evaluating LLM Adoption of Exercise Typology using Few-shot strategies.

5.5.2 Few-Shot Prompting Results

The prompt of this final architecture was tested on the same LLMs as before: ChatGPT (GPT-5), Gemini 2.5-Flash, Claude (Sonnet 4.5), and GPT-OSS. Llama 4 will not be used due to its poor performance in the last test. The evaluation process is based on the methodology described in section 5.1 and uses qualitative and quantitative metrics such as skill progression flow, feedback, exercise diversity rate, and typology alignment rate. The results obtained are presented below:

ChatGPT (GPT-5): The model adheres to the exercise typology presented in the prompt; however, the complexity of the exercises are similar to past tests. In addition, the model uses several types, such as refactoring, explaining the purpose of the code, or finding the error; but it does not present any code-writing task, which represents a problem in the learning strategy; in the same way, the model does not use any Parson puzzle or Missing Lines exercises. The GPT model proposes 17 exercises, of which 10 were part of the typology. It represents 58.82% of exercises related to fitness, which is an improvement from 23.1% of the last test.[16]

Gemini 2.5-Flash: The Gemini model is showing a real improvement with a much better variety of exercises. It is successfully generating novel exercises, including, for instance, Parson puzzles, missing lines code snippets, and actual code-writing tasks. This represents an important improvement in the adherence to the typology. In addition, it generates different types of exercise even for the most basic skills in the sensorimotor stage. The only technical point is a minor metadata error where the skill ID seems to get stuck on 'S' regardless of the actual learning stage. This model presents 18 exercises with a total of 15 exercises as part of the typology list, representing 83.33%, a great improvement compared to the last test. [17]

Claude (Sonnet 4.5): This model shows a strong performance with respect to the variety and adhesion of the exercise to the typology list. In particular, it includes writing code tasks in the Concrete Operational stage, as well as outstanding refactoring and error-finding exercises. However, in the Pre-operational stage, the model does not show Parson puzzles or missing line exercises; instead, it presents an error showing a Sensorimotor stage exercise in the Pre-operational stage. This model excels in offering more practice exercises than other models and, again, does not show the entire solution when asked. In general, this model presented a total of 23 exercises, of which 17 completely respect the typology list. This represents 73,91% of the exercises, a better performance compared to 27.8% of the last test. [18]

GPT-OSS (gpt-oss-120b): This model strictly adheres to the exercise typology, but it has some areas of improvement. In the Sensorimotor stage, the exercises feel simple, where the term-matching tasks often list the definitions in the exact same order as the terms, making the answer

[16]Complete interaction for the GPT-5 experiment can be accessed at `https://drive.google.com/file/d/1JQbRLmcYZwFUz6TeHx6yymreQNLQajQI/view?usp=sharing`

[17]Complete interaction for the Gemini 2.5-Flash experiment can be accessed at `https://drive.google.com/file/d/1k0hYHbpUpsVxyzWG4chCEnnZiqXtC5h0/view?usp=sharing`

[18]Complete interaction for the Claude Sonnet 4.5 experiment can be accessed at `https://drive.google.com/file/d/1PVWLXotYIHSLU9mwbwNX1QUy08AH76mA/view?usp=sharing`

obvious and not good for explaining definitions. On the other side, the Pre-operational stage defines good exercises, not only showing tracing examples, but also including missing line tasks. In the Concrete Operational stage, the model presents code-writing exercises with a reasonable level of complexity, some debugging examples, and a few refactoring tasks. Although the level of exercises is considered well, it would be better to have more specialized tasks like refactoring or missing error exercises. In general, this model produced a total of 17 exercises, of which 12 were strictly part of the exercise typology; it represents 70,59%, an improvement from 25% of the last experiment. [19]

In conclusion, the use of the few-shot approach in the prompt, as well as the reordering of the instructions and the strict output schema with metadata information, reflect a great improvement in the adherence of the LLMs to the exercise typology described in Section 3.2. The summary of the final percentages of adherence is shown in Table 5.4. This is an important result because this experiment shows the potential of the models to follow a learning framework, in this case the Neo-Piaget model for programming learning.

Table 5.4: Exercise Alignment with Typology Guidelines in Test 4 Prompt for Exercises Generation

Model	Total Exercises	Aligned Exercises	% Aligned
GPT-5 (ChatGPT)	17	10	58.82%
Gemini 2.5 Flash	18	15	83.33%
Claude 4.5 Sonnet	23	17	73.91%
GPT-OSS 120B	17	12	70.59%

Finally, the integration of a few-shot prompting architecture, combined with the reordering of pedagogical instructions and the enforcement of a strict, metadata output schema, established a highly constrained *Remediation Engine*. By explicitly indicating the correct LLM exercises examples, the model guaranties strict adherence to the Neo-Piagetian exercise typology detailed in Section 3.2, which was the main objective of this section. This final iteration successfully bridges the cognitive diagnosis and the practical exercises generation, which allows to have a complete autonomous system for student tutoring. Notice that it is not only generating trivial exercises, it is generating personalized practice coding exercises based on a specific cognitive complexity guided by a well-defined learning framework.

[19] Complete interaction for the GPT-OSS-120b experiment can be accessed at `https://drive.google.com/file/d/1hGCyFPuzcxoRuibirXrhm4A7iDrlv-1_/view?usp=sharing`

In summary, the experimental evaluation of the *Remediation Engine* highlights the necessity of advanced prompt engineering to enforce the alignment with the Neo-Piaget learning framework. Although baseline zero-shot approaches struggled to provide diverse and framework-aligned exercises, the implementation of a few-shot architecture successfully guided the models to generate varied and aligned programming exercises. This process successfully achieves the objective of creating a real-time adaptive learning loop, in which the model can use the initial cognitive diagnosis in order to generate appropriate programming exercises. This leads us to the third and final component of the proposed tutoring system, the long-term *Predictive Engine*.

6

The Predictive Engine: Iterative Design and Evaluation

In this section, we detail the third component in the autonomous tutoring system: the prediction engine. In this case, a knowledge tracing process was developed with the aim of predicting academic risk in C programming students, specifically the results of the first partial test at the Programming Course of the University of Padua. In this project, we proposed a hybrid architecture that merges the understanding of Large Language Models with the temporal predictive power of knowledge tracing (KT). In the first phase of the study (presented in sections above), we used a fine-tuned model to transform raw student interactions into cognitive labels based on the Neo-Piagetian framework; now, this subsequent phase leverages those rich, structured labels to model the student's learning trajectory.

By modeling these sequential interactions of students with programming exercises, the system identifies failure patterns and misconceptions of the students during their learning process. Unlike standard Knowledge Tracing models, this approach integrates the Neo-Piagetian cognitive stages, allowing the model to understand not just if a student failed an exercise but also the cognitive stage of the task students struggled with. In that way, using the learning trajectory of each student, the system is able to predict the final result of the partial exam of the programming course at the University of Padua.

6.1 Quantitative Evaluation Strategy

The main objective of the Knowledge Tracing Engine is to predict the probability of a student failing or passing the programming exam. In order to assess the predictive capabilities of the proposed neural network architectures, a quantitative evaluation framework was established.

Quantitative Evaluation Metrics: For this process, standard classification metrics such as Accuracy and F1-Score (previously defined in Section 4.1) remain foundational, but evaluating the predictive model requires some specialized metrics, described next.

- **F1-Score (Risk/Macro):** The risk F1 score for the specific "Fail" class ensures that the model accurately identifies at-risk students without excessively triggering false positives. The Macro F1-Score was also tracked to measure the overall balance between failure and pass predictions.
- **ROC Curve and AUC:** The ROC curve plots the True Positive Rate against the False Positive Rate at various threshold settings. The Area Under the Curve (AUC) was used as a primary aggregate measure of the model's ability to distinguish between students who will pass and those who will fail.
- **K-Fold Cross-Validation (5-Folds):** To ensure that the model's performance was stable and not the result of a lucky train-test split, a 5-fold cross-validation strategy was implemented. The training loss curves across all five folds were monitored to evaluate learning stability and convergence.

Attention Maps Analysis: To understand the internal decision-making process of the neural network, visual Attention Maps were generated. This technique correlates the normalized performance scores of a student over time with the specific attention weights assigned by the model. By visualizing these maps, it is possible to verify whether the model makes predictions based on the entire knowledge trajectory or if it incorrectly focuses only on the final interactions.

This evaluation process allows us to measure the performance of the predictive model, and at the same time it is useful to understand how the different architectures proposed analyze the learning trajectory to generate a final prediction.

6.2 Dataset Construction and Feature Engineering

This section details the complete data preparation pipeline, outlining how raw Moodle logs are first extracted and subsequently transformed through feature engineering to encode student interactions into sequences suitable for the LSTM network.

6.2.1 Dataset Construction

The first step of this process is to obtain the dataset used to train the predictive model. This was done using the Moodle Logs of the Programming Course of the Bachelor's Degree in Computer Science at the University of Padua, which were provided by the faculty. In order to obtain the dataset, we analyzed the logs from the year 2023-2024, as they were not used in the previous phase of the fine-tuned model (see Section 4.4).
First, a Python script was created to process the XML files into readable Excel files to identify the exercises that are useful for the task; i.e., the exercises were categorized by the topic each is treating. Secondly, to have a complete dataset that has only real information, all the students' submissions for each exercise were considered as part of the dataset; this is a similar process as the one conducted for the ground-truth dataset for the evaluation strategy of the *Diagnostic Engine*.
The next step of this dataset construction was to manually identify the skills each exercise was evaluating to add them to the exercise information. With this process, the complete dataset ends with a total of 35 exercises and 4306 student submissions, again representing a dataset of exercises with real answers, but without learning framework labels.
Consequently, all those 4306 submissions need to be tagged with the learning framework labels (Missing Skills, Mastered skills, and Cognitive Stage), which makes manual labeling unfeasible; for that reason, the fine-tuned model created in section 4.4 was used to obtain that information. The process involves passing each exercise submission to the fine-tuned model to obtain the cognitive diagnosis, which will be then used for the knowledge tracing training.
After that, the dataset was reordered to show a distribution not based on an exercise's submissions but instead, based on submissions of each student, i.e., the new focus of the dataset is the student. In that way, the dataset was modeled in order to have a time ordering of each student's submission, with a final number of 212 students who interacted with the Moodle system during that academic year; algorithm 6.1 shows an example of what the final data set looks like based on the student information.

Algorithm 6.1 Sequential Student Data Structure for Knowledge Tracing dataset

```
{
  "student_2706": [
    {
      "sequence_order": 1,
      "exercise_id": "VAR_01",
      "topic": "Variables",
      "title": "Calculate the last date expressible by a Linux system",
      "evaluated_skills": ["S1", "S2", "P1", "P2", "C1", "C2"],
      "grade": 100.0,
      "code_snippet": "#include <stdio.h>...;\n}",
      "cognitive_stage": "Concrete Operational",
      "mastered_skills": ["C1", "C2", "P1", "P2", "S1", "S2"],
      "missing_skills": []
    },
    {
      "sequence_order": 2,
      "exercise_id": "COND_01",
      "topic": "Conditionals",
      "title": "Determine if a year is a leap year",
      "evaluated_skills": ["S1", "S2", "P1", "P3", "C1", "C2", "C4"],
      "grade": 100.0,
      "code_snippet": "#include <stdio.h>\...",
      "cognitive_stage": "Concrete Operational",
      "mastered_skills": ["C1", "C2", "C4", "P1", "P3", "S1", "S2"],
      "missing_skills": []
    }
  ]
}
```

Once we have the sequence of exercises for the students, the next step is to define the target variable, which in this case is the information about the final grade for the first partial exam of the student. This grade is then transformed into a qualitative variable: PASSED(0) or NOT PASSED (1), depending on whether the grade is equal to or greater than 18, which is the standard value of the University of Padua.

In order to get the target value, it was necessary to match the student id from the Moodle Logs with the university's student id by using the file *users.xml* provided in the logs. However, during

this process, three scenarios were found:

- Some of the students have a final grade for the Programming Course, but they did not interact with the Moodle exercises, so it is not possible to have a learning interactions timeline.
- Some students interact with Moodle exercises, but they do not have a final grade, so they are not considered in the final dataset.
- Most of the students interact with the Moodle exercises and have a final grade; as a consequence, we have learning interactions. These students are the ones to be considered in the final dataset.

Taking these scenarios into account, the final dataset is composed of a total of 175 students with 3815 submissions.

6.2.2 Dataset Encoding

The final dataset must be encoded to be used in the training process; for this reason, some strategies were tried.

The first attempt for the encoding was to create a multi-hot encoding for all the skills of each topic and a one-hot encoding for the cognitive stage. A schematic graphic is presented in Table 6.1. As can be seen, the skills for each topic were encoded with a name as: *TOPIC_SKILL_ID*; thus, the IDs of the skills are different for each topic, which could help the model identify the difference between them. After that, to identify missing and mastered skills, the defined encoding was (+1) for mastered skills, (-1) for missing skills, and (0) for not assessed. It is important to note that each interaction between the students at time t represents an exercise on a specific topic. Consequently, each array of data at time t will have some columns with values of +1 or -1, and the rest of the columns will be 0. If a student did not do a specific exercise, the time t for that exercise will be 0. Finally, the one-hot encoding of the cognitive stage creates 3 columns to identify whether the exercise at time t was categorized as Sensorimotor, Pre-operational, or Concrete Operational.

Finally, the dataset ends with 105 columns, representing very sparse input vectors, which creates a sparsity problem given the small number of students (175) and a large number of features (105). This represents a problem for the training process because for each iteration, the model will analyze just a few skills depending on the topic, and all the rest of the features will be analyzed as 0.

Table 6.1: First Experiment of Encoding for the Dataset for the Knowledge Tracing Process. Variables are abbreviated for formatting constraints (e.g., C_S1 = COND_S1, Sens. = isSensorimotor).

ID	Exercise	T	C_S1	C_S2	C_P1	C_C1	L_S1	L_P1	L_C1	Sens.	Pre.	Conc.
1	COND_01	1	1	-1	1	-1	0	0	0	1	0	0
1	LOOP_01	2	0	0	0	0	1	1	-1	0	1	0
1	-	3	0	0	0	0	0	0	0	0	0	0
2	COND_01	1	1	1	1	1	0	0	0	0	0	1
2	COND_02	2	1	1	1	1	0	0	0	0	0	1
2	LOOP_01	3	1	1	1	1	0	0	0	0	0	1

Considering those limitations, a second encoding strategy is proposed in order to solve the high-dimensionality problem by not using all the skills for each topic. This is done due to the general idea that the model needs to know where the student failed and in which cognitive stage. For that reason, the new encoding proposes a normalized score for each specific topic (see Table 6.2), in which, for each time step, the score is calculated based on the ratio of mastered (+1) vs. missing (-1) skills. In that way, we have a final value that encapsulates the information about the skills for each topic. Finally, for the cognitive stage, instead of a one-hot encoding, it generates just one column with a normalized scalar defined as: 0 for Sensorimotor, 0.5 for Pre-operational, and 1 for Cognitive Stage.

Table 6.2: Second Experiment of Encoding for the dataset for the Knowledge Tracing Process. Variables are abbreviated to fit formatting constraints (e.g., EID = EXERCISE_ID, LEXS = LOGEX_SCORE, STG = STAGE).

SID	EID	T	EXPS	LEXS	VARS	CONDS	LOOPS	FUNCS	PNTS	ARRS	STG
1	COND_01	1	0	0	0	0.55	0	0	0	0	0.5
1	LOOP_01	2	0	0	0	0	0.78	0	0	0	1
1	-	3	0	0	0	0	0	0	0	0	0
2	COND_01	1	0	0	0	0.55	0	0	0	0	0.5
2	COND_02	2	0	0	0	0.55	0	0	0	0	0.5
2	LOOP_01	3	0	0	0	0	0.78	0	0	0	1

Finally, using this encoding, the dataset has only 9 columns, which eliminates the high-dimensionality problem, and at the same time, it captures the skills information using scores.

6.3 Knowledge Tracing Training

The training process of the knowledge Tracing model must consider the temporal dynamics of the student information, as their exercise interactions are sequential. As proposed by Piech

et al. [17], the use of deep neural networks can capture more complex representations of student knowledge, resulting in substantial improvements in prediction performance. For that reason, an LSTM (Long Short-Term Memory) model is used as the main method for the training process. This process considers some different experiments to get a higher performance of the model.

6.3.1 Training using Large Dimensionality Encoding:

As described in the previous section, the first encoding method has a total of 105 dimensions and was trained using a baseline model with a standard Unidirectional LSTM, which has a hidden dimension of 64 neurons and a single layer. The model was trained using Adam optimizer with a learning rate of $3e^{-3}$ and a weight decay of $1e^{-3}$ for regularization. Given the limited size of the dataset ($N = 175$) with respect to the high feature space (105 features), a higher regularization technique was necessary to mitigate overfitting. For that reason, we applied a dropout rate of 0.45 to the LSTM output.
In order to have the best approach for testing the dataset, instead of splitting the dataset into training and test sets, in this process, a 5-Fold Cross-Validation scheme was used to ensure that all student subgroups were represented in the validation sets. In addition, the loss function was a weighted Binary Cross-Entropy Loss (BCEWithLogitsLoss) to obtain dynamically calculated class weights based on the ratio of failed-to-passed students in each training fold. In this way, the model is not obligated to always calculate the same ratio for distinguishing a PASS or NOT PASSED class; instead, it adapts to the current dataset.
The evaluation of this first strategy revealed several limitations inherent to this encoding approach and its very sparse vectors. The model demonstrated a "Curse of Dimensionality" problem, achieving an average Accuracy of 64.35% and an F1-Score (Risk) of 63.94%. These results demonstrated a high variance between folds, as shown in Figure 6.1, indicating instability of the model and the high difficulty of learning robust patterns.

6.3.2 Training using Normalized Scores Encoding:

Given the limitations of sparse input vectors, as described earlier, the second encoding method uses only 9 dimensions (9 topic scores + 1 Cognitive Stage) to address the Curse of Dimensionality. As was defined in the previous architecture, this model also uses a standard unidirectional LSTM, but this time adding an Attention layer.
This model has a hidden dimension of 64 neurons and a single layer. Then, an attention layer

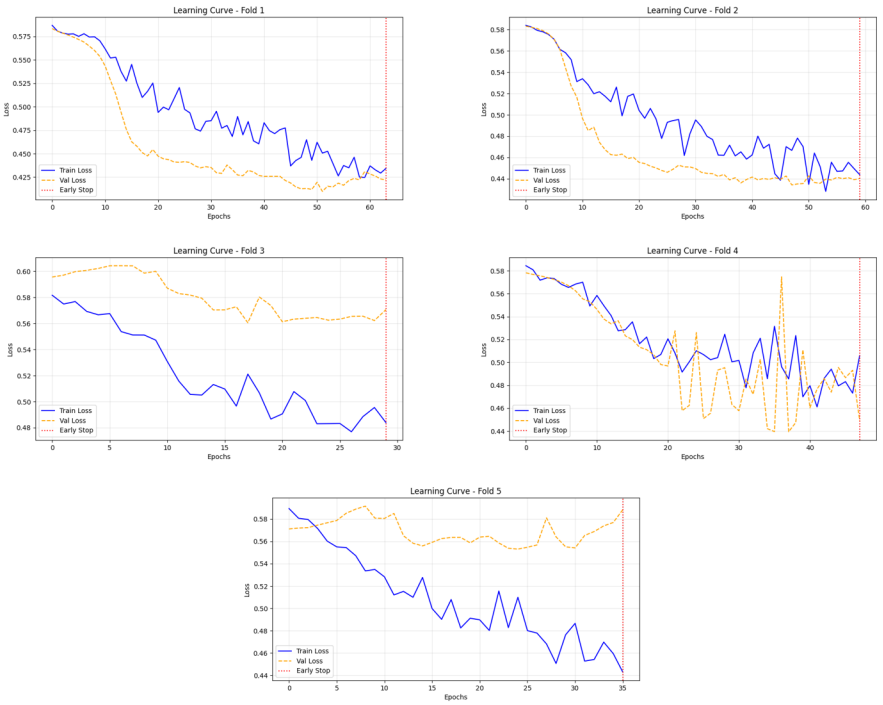

Figure 6.1: Loss Curves of 5-Folds for the Knowledge Tracing Model using Large Dimensionality Encoding.

computes a score for each time step in the sequence. These scores are normalized using a softmax function to generate attention weights. Unlike the baseline model, which used only the final hidden state, the attention layer computes a vector as the weighted sum of all hidden states, allowing the model to focus on specific critical exercises in the student interactions.

The training parameters were adapted to the new architecture, in which the model uses Adam optimizer with a learning rate of $5e^{-3}$ and a weight decay of $1e^{-3}$ for regularization. Given that the model has a low feature space, a lower regularization value was used; in this case, we applied a dropout rate of 0.2 to the LSTM output.

Similarly to the last proposed architecture, instead of splitting the dataset into training and test sets, a 5-Fold Cross-Validation scheme was used to ensure that all student subgroups were represented in the validation sets. The loss function was a weighted Binary Cross-Entropy Loss (BCEWithLogitsLoss) to obtain dynamically calculated class weights based on the ratio of failed-to-passed students in each training fold. In this way, the model is not obligated to always calculate the same ratio for distinguishing a PASS or NOT PASSED class; instead, it adapts to the current dataset. This is with the aim of finding the optimal ratio that maximizes the F1-Score for the Risk Class.

After the training process with this encoding approach, the results of this architecture improve over those obtained in the baseline. The model achieved an average Accuracy of 74.29% and an F1-Score (Risk) of 74.65%, which can be evidenced in the confusion matrix and the ROC curve of Figures 6.2 and 6.3. It indicates that the model performs well in predicting whether the student will pass or not the exam, missing only 15 students as false positive values. In the same way, the 5-fold loss curve shows a more stable training; however, it continues to show some peaks (see Figure 6.4).

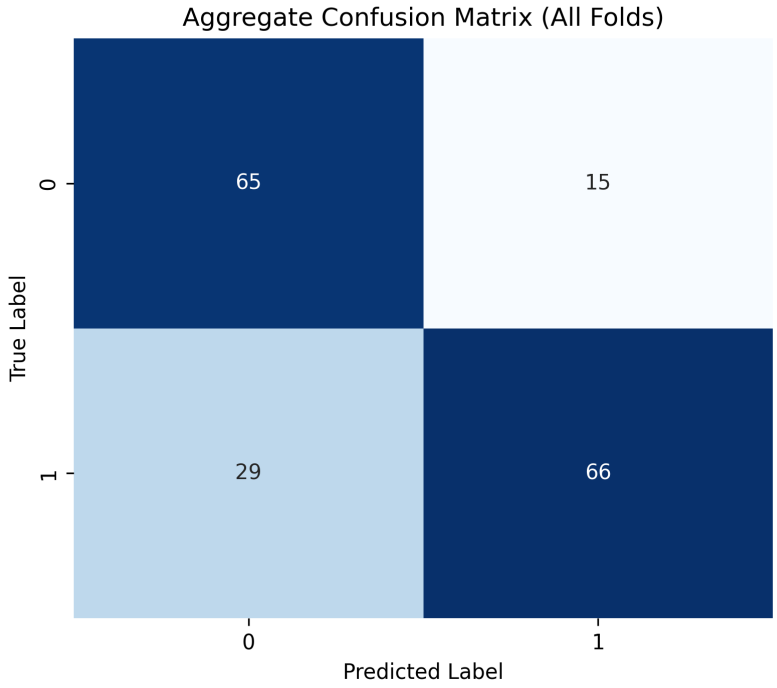

Figure 6.2: Confusion Matrix for the Knowledge Tracing Model using LSTM with Attention architecture.

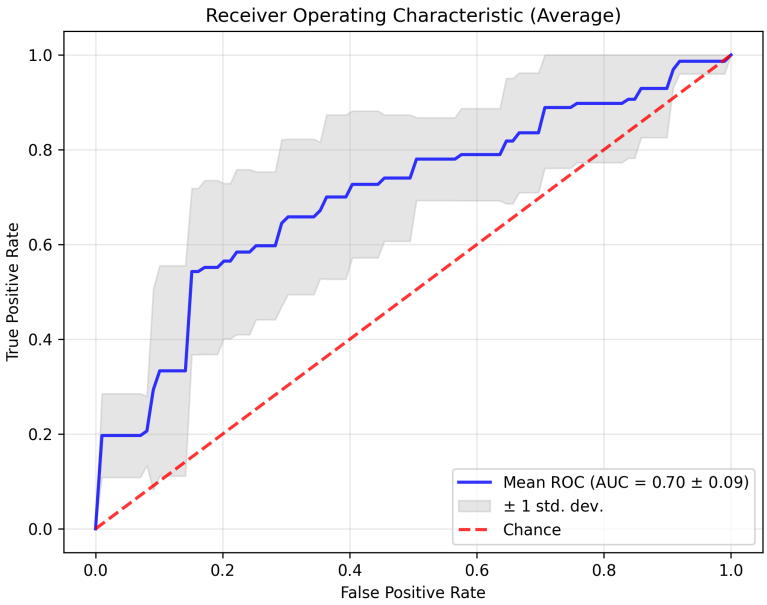

Figure 6.3: ROC Curve for the Knowledge Tracing Model using LSTM with Attention architecture.

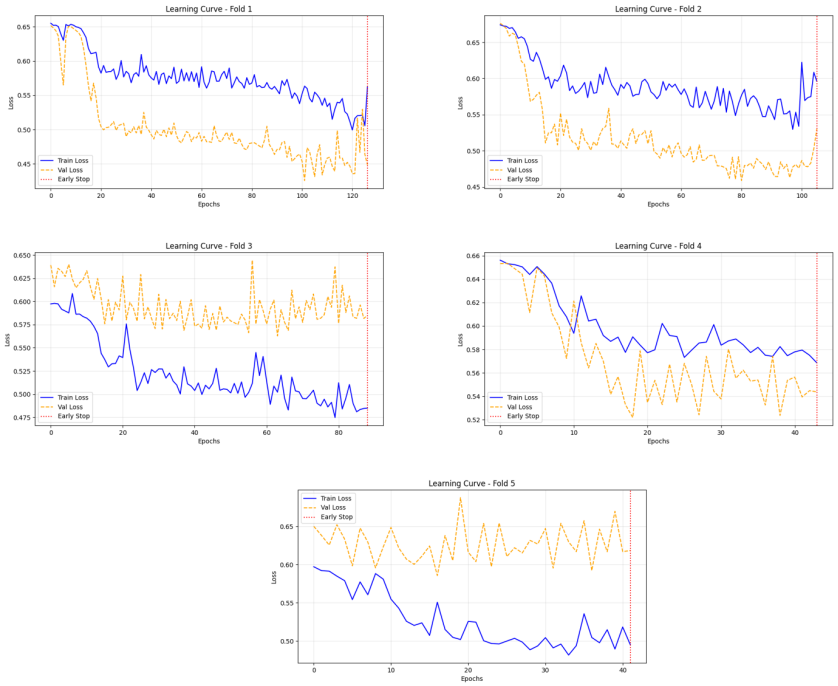

Figure 6.4: Loss Curves of 5-Folds for the Knowledge Tracing Model using LSTM with Attention architecture.

In order to understand the decision made by the model, we analyzed the Attention Maps of specific student examples. These graphics produce a correlation between the amount of attention (weights) applied to specific exercises of the student, which are represented as blue bars, and the normalized scores of each exercise (represented in the red line). For instance, Figure 6.5 shows a false positive case in which the student failed the course (True: 0.0) but was predicted to pass with a Prediction value of 0.75. The red line, which represents the performance, shows a low value in the Array topic at the end of the course interactions, and the blue bars indicate that the model paid more attention to the final exercises of the course, which is the reason why it predicts a failure of the exam.

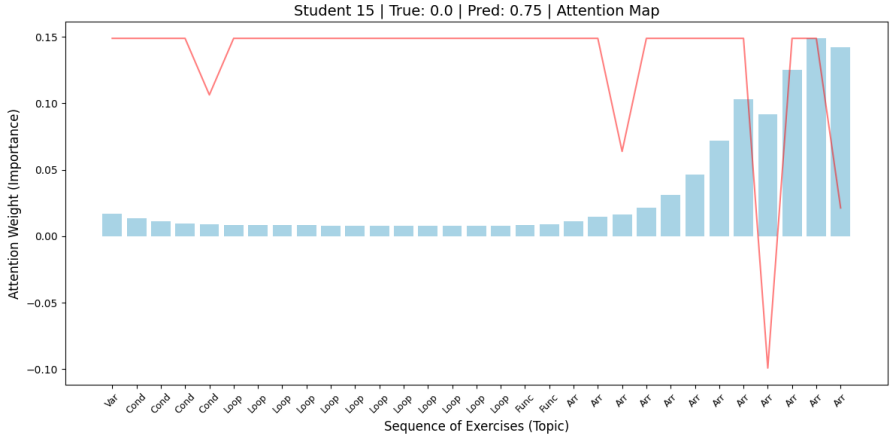

Figure 6.5: Attention Map - False Positive for the Knowledge Tracing Model using LSTM with Attention architecture.

Another example is shown in Figure 6.6, where we present a true risk prediction, where the student failed (True: 0.0) and was correctly flagged by the model (Pred: 0.42). This student shows a learning trajectory with some errors in various topics, which helps the model not to focus only on the last interactions.

In conclusion, the use of the normalized score encoding helps the model to learn more complex patterns due to the dense vector values. In addition, the LSTM with attention model generates correct predictions with a good accuracy value; however, as shown in previous examples, it is necessary for the model not to focus just on the end of the interactions; instead, for the task the model is performing, it is necessary to apply the attention mechanism to the entire knowledge trajectory, so the predictive model can understand that an error at the initial topics can also generate a risk of failing the final exam.

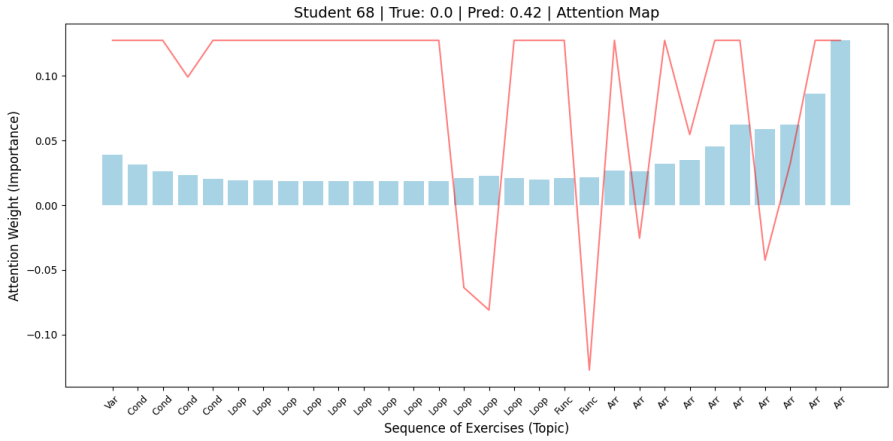

Figure 6.6: Attention Map - True Negative for the Knowledge Tracing Model using LSTM with Attention architecture.

6.3.3 Training using Bi-LSTM with Attention:

As described before, it was necessary to force the model to pay attention to the whole knowledge tracing to enhance the model's ability to capture complex dependencies. For that reason, a Bi-LSTM (Bidirectional LSTM) was proposed using the same encoding strategy. This model has a hidden dimension of 64 neurons and a single layer, but this time it is executed twice due to the bidirectionality, having a total of 128 neurons. Then, the Attention Mechanism was adapted to process this enriched context vector, which allows the model to weigh the importance of an exercise based on its relationship with both the student's history and their future outcomes.
The training parameters were adapted to the new architecture, in which the model uses Adam optimizer with a learning rate of $4.5e^{-3}$ and a weight decay of $1.2e^{-3}$ for regularization. In this case, we applied a slightly higher dropout rate of 0.3 to the LSTM output. In addition, it is noticeable that a persistent challenge is the limited dataset size, which makes the model tend to overfit. To address this, we implemented a Gaussian Noise Injection strategy where random noise sampled from a normal distribution was added to the input vectors in order to increase the original dataset.
Similarly to the previous architectures, instead of splitting the dataset into training and test sets, a 5-Fold Cross-Validation scheme was used to ensure that all student subgroups were represented in the validation sets. The loss function was a weighted Binary Cross-Entropy Loss (BCEWithLogitsLoss) to obtain dynamically calculated class weights based on the ratio of failed-to-passed students in each training fold. In this way, the model is not obligated to always

calculate the same ratio for distinguishing a PASS or NOT PASSED class; instead, it adapts to the current dataset. This is in order to find the optimal ratio that maximizes the F1-Score for the Risk Class.

Based on this definition, after the training process, the results of this architecture improve over those obtained in the last attempt. The model achieved an average Accuracy of 76% and an F1-Score (Risk) of 75.88%, which can be evidenced in the confusion matrix and the ROC curve of Figures 6.7 and 6.8. It indicates that the model performs well in predicting whether the student will pass or not the exam, missing only 13 students as false positive values. In the same way, the 5-fold loss curve shows a more stable training (see Figure 6.9).

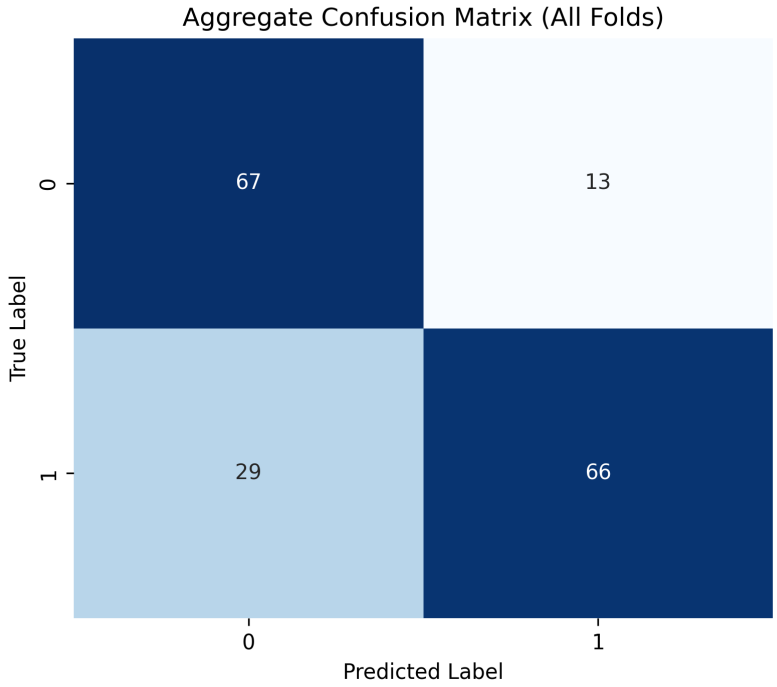

Figure 6.7: Confusion Matrix for the Knowledge Tracing Model using Bi-LSTM with Attention architecture.

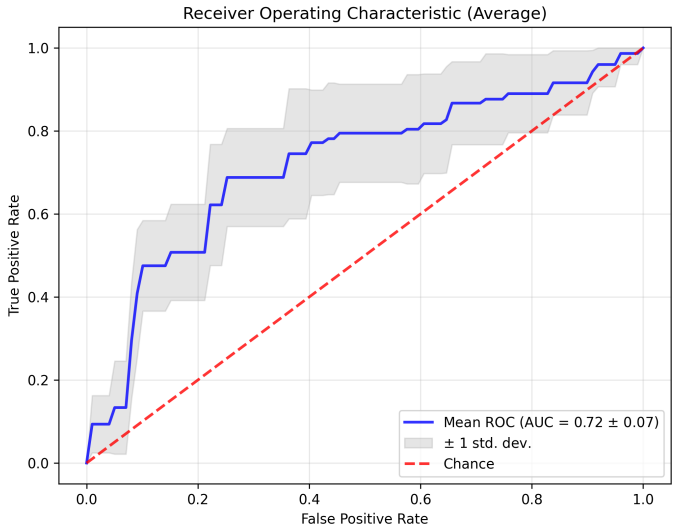

Figure 6.8: ROC Curve for the Knowledge Tracing Model using Bi-LSTM with Attention architecture.

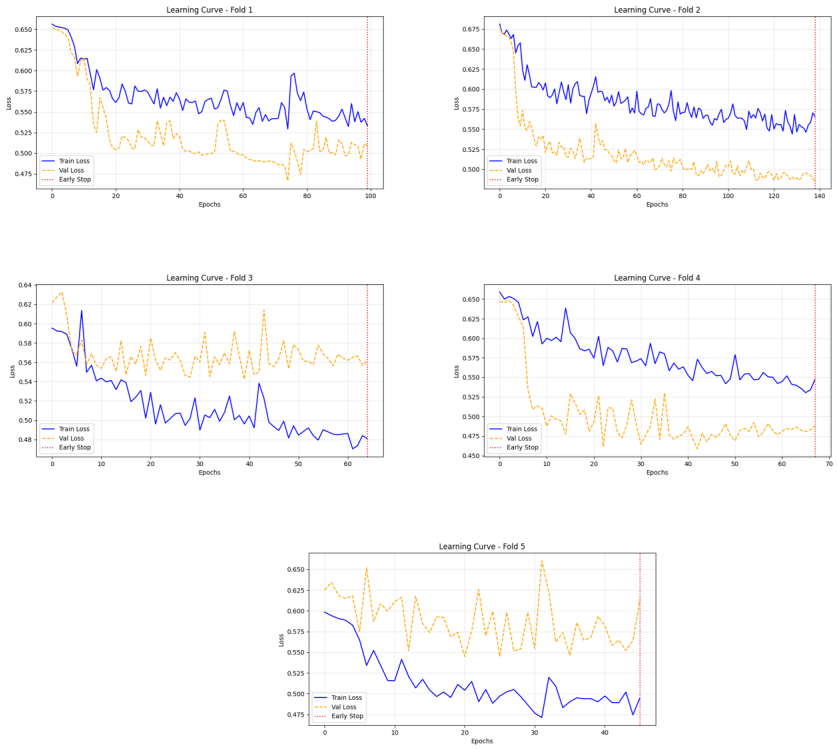

Figure 6.9: Loss Curves of 5-Folds for the Knowledge Tracing Model using Bi-LSTM with Attention architecture.

As in the previous attempt, it is important to understand the decision made by the model; we analyzed the Attention Maps of specific student examples. These graphics produce a correlation between the amount of attention (weights) applied to specific exercises of the student, which are represented as blue bars, and the normalized scores of each exercise (represented in the red line). For instance, Figure 6.10 shows a true positive case in which the student passed the course (True: 1.0) and it was predicted to pass with a Prediction value of 0.78. The red line, which represents the performance, shows a few low values during the course interactions, but this time, the blue bars indicate that the model paid attention not only at the end of the trajectory, but it also has a more stable attention mechanism.

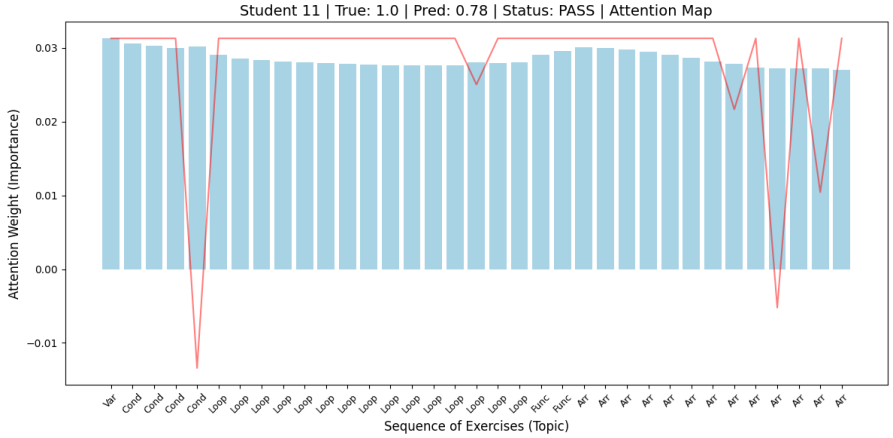

Figure 6.10: Attention Map - True Positive for the Knowledge Tracing Model using Bi-LSTM with Attention architecture.

Another example is shown in Figure 6.11, where we present a true risk prediction, where the student failed (True: 0.0) and was correctly flagged by the model (Pred: 0.33). This student shows a learning trajectory with some errors in various topics; the model paid attention to the whole sequence and helped to detect the student's misconceptions.

In conclusion, these experiments define the Bi-LSTM with Attention as the superior architecture for predicting academic risk using C code submissions as a knowledge trajectory. The use of attention in both directions of the network, in combination with regularization techniques and dynamic threshold tuning, allows the model to understand complex patterns of how students learn.

In order to improve the results obtained by the Bi-LSTM architecture, other experiments were tried using Gated Recurrent Unit (GRU) with Attention. However, the results obtained were very similar and do not reveal a significant improvement; particularly, the GRU approach ob-

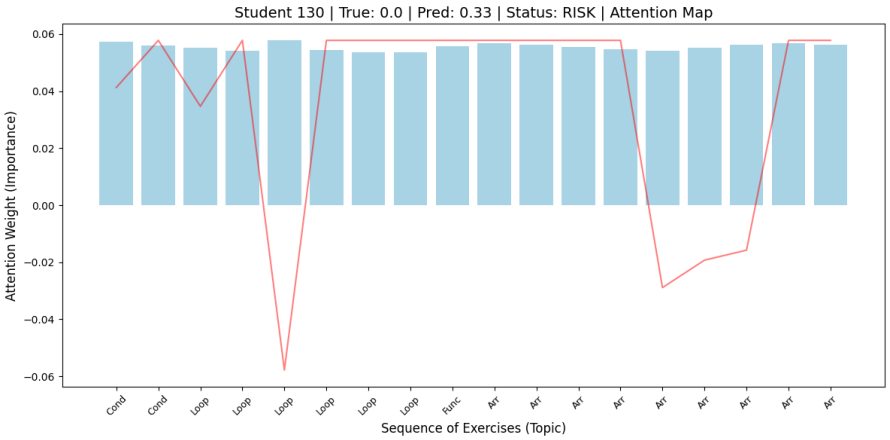

Figure 6.11: Attention Map - True Negative for the Knowledge Tracing Model using Bi-LSTM with Attention architecture.

tained an average Accuracy of 76.57% and an F1-Score (Risk) of 75.77%; figure 6.12 shows the ROC curve of this approach. In general, the similar performance between LSTM and GRU suggests that the predictive ceiling is constrained by the size of the dataset.

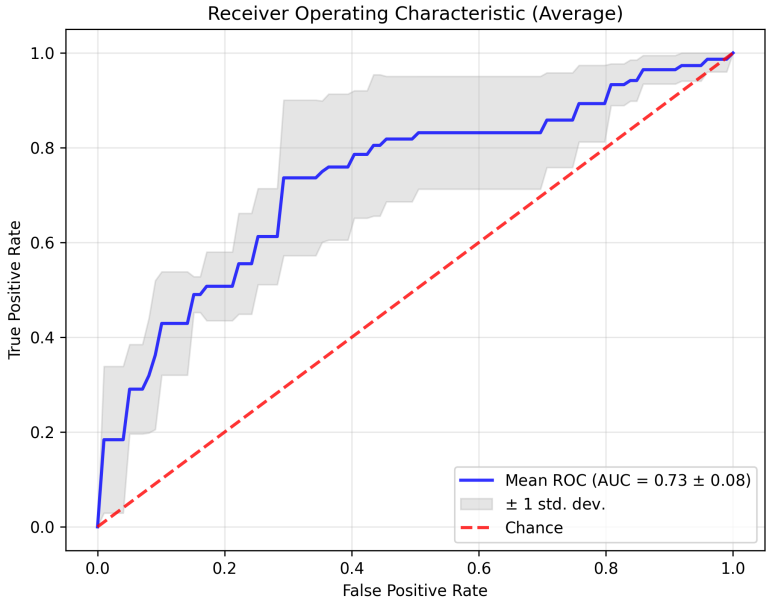

Figure 6.12: ROC Curve for the Knowledge Tracing Model using GRU with Attention architecture.

In conclusion, the LSTM approaches that were part of these experiments got relevant results

for the knowledge tracing model. It was also because of the use of the Neo-Piaget model, which indicates not only if the student passed or not an exercise but also at what level of understanding the student is. Finally, the enriched labels of the learning framework give the knowledge tracing model more information, which is useful to determine the final risk of a student in a programming course.

In order to support these results, the same architecture (Bi-LSTM with attention) was used on the same dataset, but using the standard knowledge tracing strategy, i.e., not using the enriched labels of the Neo-Piaget framework. This experiment used only the information of passing or not passing the exercise of each student, based on the grade obtained (100 points means that the student passed the exercise). The results of this baseline experiment, using the same parameters as before, obtained an average Accuracy of 67.43% and an F1-Score (Risk) of 58.59%, which can be evidenced in the confusion matrix and the ROC curve of Figures 6.13 and 6.14. It indicates that the model is not performing well by using only basic labels for early risk prediction.

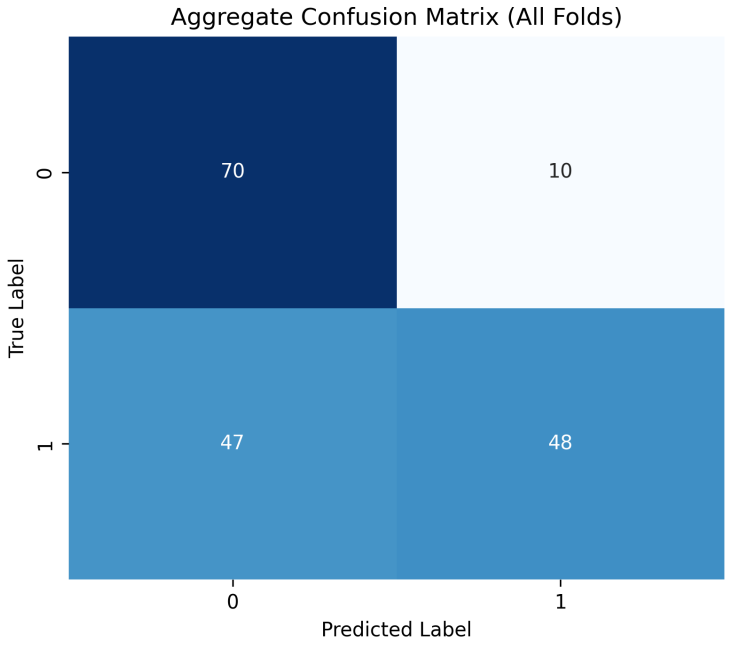

Figure 6.13: Confusion Matrix for the Knowledge Tracing Model using baseline without enhanced labels.

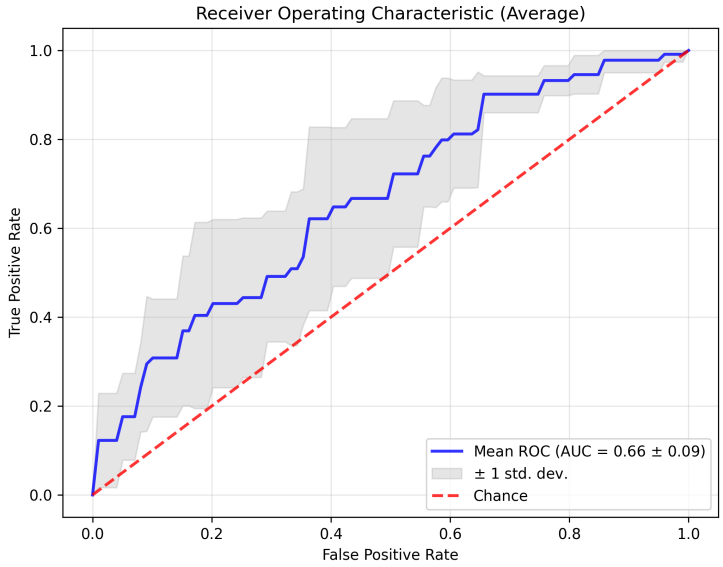

Figure 6.14: ROC Curve for the Knowledge Tracing Model using baseline without enhanced labels.

Finally, Table 6.3 compares the final results obtained by Bi-LSTM with Attention architecture and the traditional baseline for Knowledge Tracing. These results confirm that the use of enriched labels obtained by the LLM based on the Neo-Piaget framework outperforms the basic Knowledge Tracing neural network technique.

Table 6.3: Performance comparison of Deep Knowledge Tracing using BiLSTM with Attention, with and without enriched labels.

Metric	Baseline (No Enriched Labels)	Proposed Model (Enriched Labels)
Accuracy	0.67	0.76
AUC	0.67	0.72
F1 (Risk/Fail)	0.71	0.76
F1 (Macro)	0.59	0.74

In summary, the design of the Bi-LSTM with Attention architecture establishes a robust predictive mechanism for the Knowledge Tracing engine. By moving beyond traditional metrics—which merely track whether a student passes or fails—to integrating the enriched Neo-Piagetian cognitive stages, the predictive model achieves a much deeper temporal understanding of the student's learning trajectory. Consequently, the system does not simply track

past performance; it effectively calculates the probable risk of academic failure in future evaluations. This predictive capability successfully closes the adaptive learning loop, transforming the tutoring system into a proactive early-warning tool for educators.

The comparative analysis presented in Table 6.3 supports this hypothesis. The results confirm that the use of this advanced architecture along with the cognitive labels generated by the fine-tuned LLM significantly outperforms the traditional Knowledge Tracing baselines. It is clear that the use of comprehensive learning states—encompassing the cognitive stage, as well as missing and mastered skills—provides the proposed *Predictive Engine* with the necessary depth to offer a highly accurate and robust mechanism for early academic risk detection.

However, while these quantitative experiments validate the accuracy and theoretical alignment of the predictive and diagnostic models, the true efficacy of the overarching tutoring system must be measured through practical application. To fully assess the usability and general student perception of this platform, the integrated system was deployed in a live academic environment. Accordingly, the following chapter presents the evaluation of the pilot deployment in a real programming course, analyzing system viability and student feedback.

7

Web Application Pilot Deployment

With the defined architecture and individual components fully integrated, the final step involved transitioning the platform to a live academic setting. In order to evaluate the end-to-end feasibility and stability of the autonomous tutoring system, the web application was deployed as a pilot study within a real introductory C programming course part of the Department of Mathematics of the University of Padua. The primary objective of this deployment was to observe the system's capacity to handle LLM-based cognitive diagnosis and real-time generation of tailored exercises.

The web application is available through the following link: `https://llmeducation-900c9.web.app/`. Students enrolled in the course, along with the professor and a teaching assistant, were provided access to the platform to practice programming concepts in a voluntary way. Given its nature, the evaluation methodology prioritized passive data collection, which means that the system was configured to automatically record detailed interaction logs in the database. The purpose of this process is to analyze the behavior of the entire tutoring system when students use the methodological components: the *Diagnosis Engine* and the *Remediation Engine*. Instead, for professors, their dashboard allows them to analyze the outcomes of the predictive model.

The pilot deployment of the proposed tutoring system gave some insights about its practical application in a real educational environment. The web application was proposed as a volunteer option to practice C programming exercises in the Bachelor's degree programming course at the Department of Mathematics in the University of Padua. During the evaluation period,

the platform successfully registered 35 students and 1 professor. In order to evaluate the stability of the application, all interactions between users and the AI model were registered in a Firestore database, in which the system successfully processed 198 total interactions without critical failures. These interactions include the entire pipeline: from the initial submission of a C code for a general exercise, going through the cognitive diagnosis made by the fine-tuned LLM, to the generation of tailored remedial exercises.

In addition to recording data interactions in the database, the pilot deployment also validated the effectiveness of the platform as an analytics tool for professors. As presented in the methodology section, a dedicated Professor Dashboard was developed to provide real-time visibility into each student's learning progression, as well as the general metrics the class is generating. The interface presented in Figure 7.1 presents a section with the general distribution of all students through the class topics and their risk of failing the final exam. Furthermore, the dashboard allows the professor to have read-only access to the detailed history of every individual student interaction.

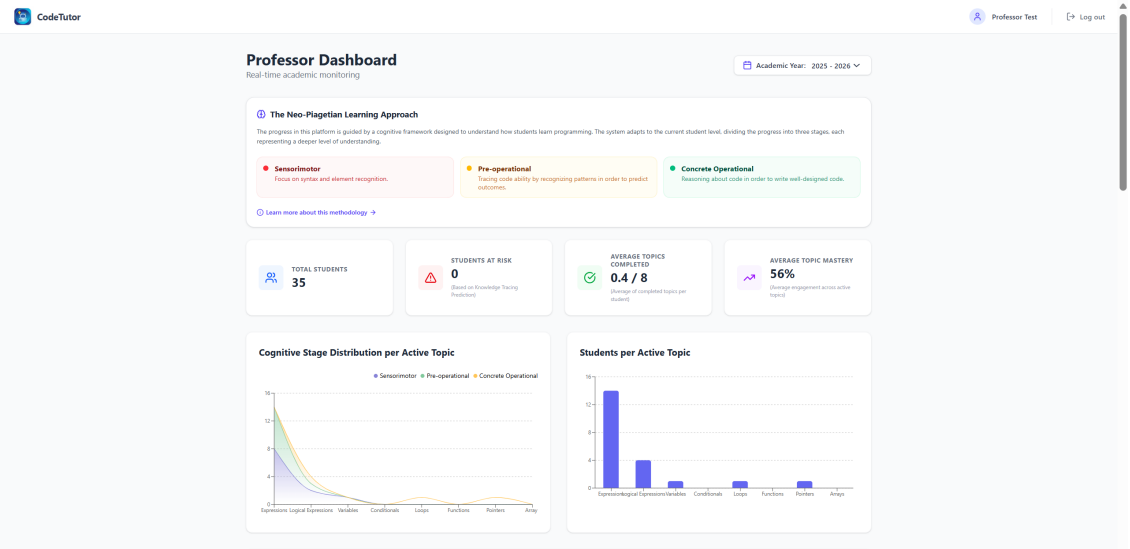

Figure 7.1: Final Professor Dashboard in the pilot deployment.

This feature allows the professor to monitor all the learning path and possible errors the platform could have. Using it, several case studies on how students interact with the system could be observed. For instance, as shown in the first part of Figure 7.2, a student starts the topic of Logical Expressions with a wrong answer in the first diagnosis exercise, which generates the fine-tuned model to classify that student into the sensorimotor stage. After that, the student continues doing the tailored exercises until he or she reaches the concrete operational stage,

which generates interesting exercises about refactoring. This represents a successful learning trajectory, carrying the student from the very basics of the C topic to master complex skills like methods refactoring.

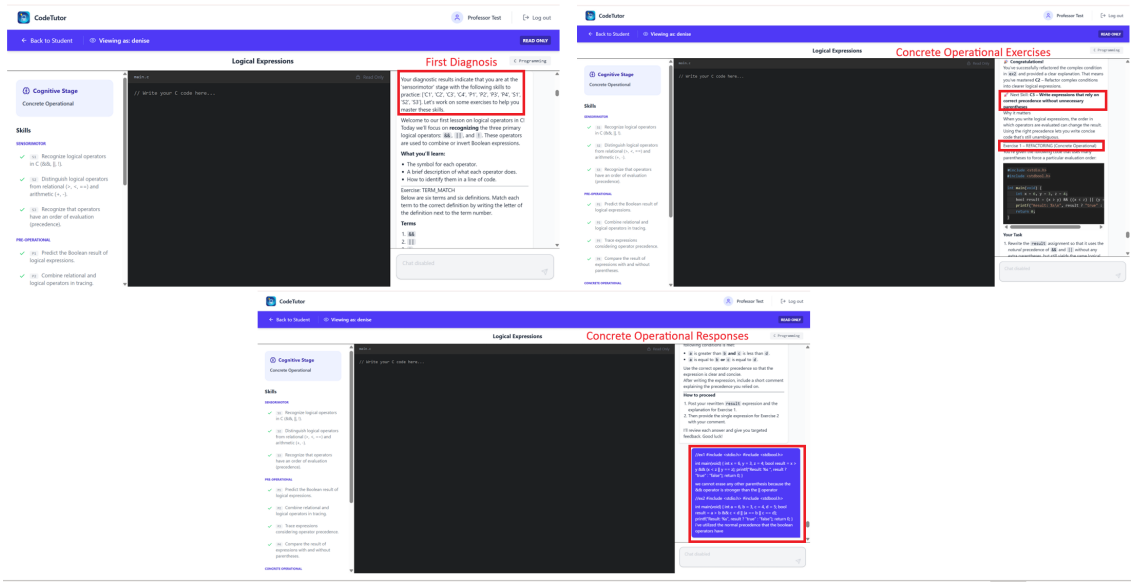

Figure 7.2: Successful Learning Interactions of a student in the Logical Expressions topic.

The presented pattern is similar to the rest of the students who interacted with the system through the first C programming topics. However, there is also a study case in which a student was classified in the sensorimotor stage and, instead of continuing the learning trajectory, the student asked for more complex exercises, trying to skip the pre-defined cognitive stages. The interaction of this case is shown in Figure 7.3. From this example, we can deduce that even when the fine-tuned model did a good job of classifying the initial cognitive stage, some students can believe that they have already understood the selected skills, and they want to go further through the topics. It is specially distinguished in the first topics (Expressions and Learning Expressions), as they are simple topics compared to arrays or pointers sections.

Finally, these real interactions validate the end-to-end feasibility of the autonomous tutoring system. As was presented in the successful case studies, it is demonstrated that the complete pipeline can effectively guide a student through the Neo-Piagetian cognitive stages, dynamically adjusting the tailored exercises from simple syntax corrections to complex refactoring tasks. However, the cases in which students attempted to bypass the natural progression of the learning framework highlight the difficult aspect of human-computer interaction, in which, while the AI successfully enforces the rigor of the framework, the learner can represent an un-

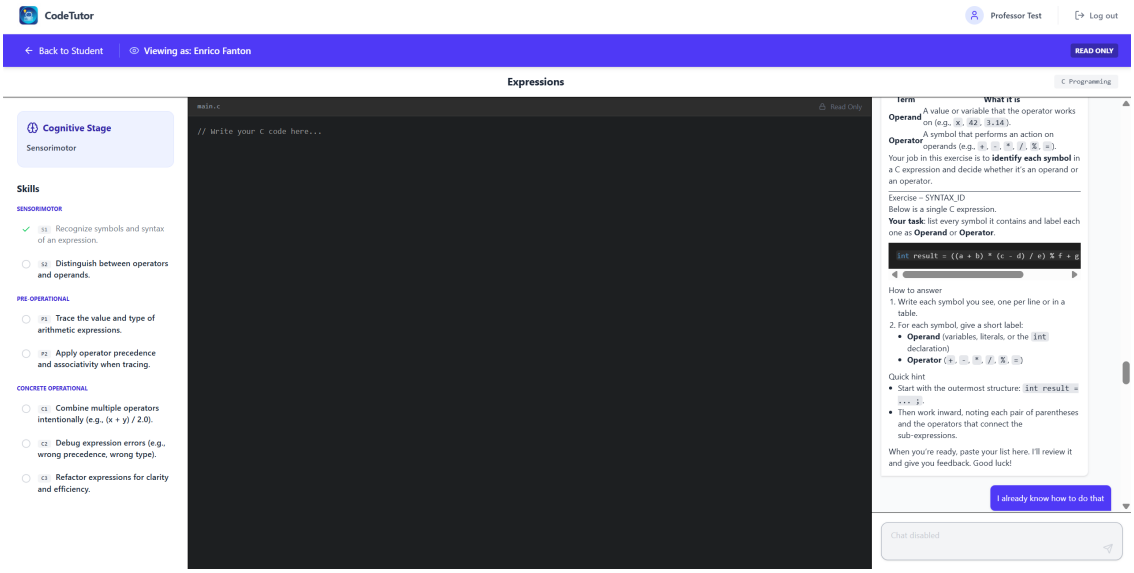

Figure 7.3: Inconvenient in Learning Interactions of a student in the Expressions topic.

predictable variable. This is the reason why a professor dashboard is useful to evaluate how good the interactions between the students and the model are working, representing a helpful input to know topics to enforce in regular classes. Overall, this pilot deployment confirms that the platform is capable of generating complete learning paths by generating rich learning analytics based on a well-defined learning framework.

8 Discussion

The previous chapters presented the methodology and results obtained for the proposed tutoring system, through its diagnostic, remediation, and predictive components. This chapter interprets the empirical results to address the research questions that drive this project. The following sections summarize the implications found by integrating the Neo-Piaget learning framework into a complete learning cycle for introductory students of C programming.

8.1 The *Diagnostic Engine*: The Feasibility of Cognitive Diagnosis

The primary objective of the *Diagnostic Engine* was to identify the cognitive stage of the Neo-Piaget learning framework, along with the missing and mastered skills from a student's raw code submission. It was performed through an iterative process in which an open-weight fine-tuned model was developed; specifically, the GPT-OSS-20b model was trained over thousands of real students' exercises from Moodle Logs. As detailed in section 4.4, the base model GPT-OSS-20b achieved an accuracy of 66% and a F1-score of 71% between all tested C programming topics; instead, the fine-tuned model achieved a total accuracy of 74% and a final F1-score values of 75% among all topics, which represents higher values compared to commercial general-purpose LLMs like Gemini 2.5-Flash which achieved an accuracy of 72%, demonstrating the specialization outcome of the fine-tuned model.

These results are significant because they answer the first research question, as they demonstrate that it is possible to use just raw exercises submissions to measure the level of understanding of C programming concepts based on a well-defined learning framework. These results indicate that an LLM can extract more information from exercises than its correctness, allowing educational settings to have more complete reasoning information about students learning. In addition, the increased accuracy and F1-score values demonstrate that it is possible that a fine-tuned model, based on a relatively limited model such as GPT-OSS-20b, could achieve better performance even compared to models with higher reasoning capabilities like Gemini 2.5-Flash.
However, these values are influenced by the natural data imbalance inherent in real-world academic logs. This is because the initial dataset was not complete enough to cover all the possible complexity scenarios among all the C programming topics. For instance, the lack of Moodle submissions with very basic syntax errors in the first topics like Expressions and logic expressions demonstrates that most of the students feel confident with those topics; this differs from more advanced topics such as pointers or arrays, where the exercise submissions have a more varied type of error between all the Neo-Piaget cognitive stages. This scenario directly affected the quality of the LLM fine tuning process, which tended to make a better diagnose in more complicated topics.
It is also notable that this proposed model is small enough to run as part of a cluster with reasonable GPU capabilities, generating a model that does not depend on costly public APIs to be executed. In the same way, since it is running locally, the model does not have to deal with data privacy constraints as if it were using public commercial models.
Different studies demonstrated the efficacy of using the Neo-Piaget framework in programming education, especially in categorization of programming errors [12] [14]. Although this existing literature highlights the importance of classifying these errors into cognitive stages, this project uses the diagnosis obtained from the LLM to create an initial point for the educational loop, which treats each cognitive stage as a barrier the student needs to overcome. For that reason, the *Diagnostic Engine* presented is the first component that interacts in the final proposed system.

8.2 The *Remediation Engine*: LLMs and Learning Framework Constraints

The purpose of the *Remediation Engine* is to generate adaptive tailored code programming exercises based on the evaluation performed by the *Diagnostic Engine*. This second step of the tutoring system was built using an iterative process based on prompt engineering techniques, in which a final few-shot prompt was developed. This prompt asks the LLM to generate varied and complex exercises following the Neo-Piaget learning framework; in order to do the task, the prompt uses a list of pre-defined types of exercises that follow exactly the methodology proposed by the framework.

As detailed in section 5.5.2, the use of this list of types of exercises helped the models understand what we expect as complex exercises, not only as C programming skills, but also to be coherent to the Neo-Piaget framework. The models with a higher adherence to the framework were Gemini 2.5-Flash and Claude Sonnet 4.5 with 83.33% and 73.91% respectively. Regarding non-commercial models like GPT-OSS-20b, prompt engineering techniques increased its alignment percentage to 70.59%, although it is significantly lower than Gemini 2.5-Flash, this model has an advantage in terms of deployment due to its possibility to run locally.

These results are relevant for the study of the second research question, which evaluates the capacity of LLMs to generate exercises based on the Neo-Piaget diagnosis and a list of specific types of exercises. For instance, during the first experiments, the only use of explanations of the Neo-Piaget stages did not enforce the model to generate adequate exercises for each one of them. After that, the use of the exercise typology list helped guide the models on which exercises are more interesting depending on the cognitive stage; however, there were cases where the models started to hallucinate. For that reason, the use of few-shot prompt techniques taught the LLMs how to generate different C programming exercises based on the presented list.

Some studies present AI-based systems that act as assistants to learn a new programming language or present the effectiveness of LLMs to generate ready-to-use programming exercises with a certain level of novelty [35]. For instance, [36] created a novel system based on constructivism learning theory and cognitive load theory, in which the authors generate exercises that divide learning into these phases. Similarly, our project uses the Neo-Piaget learning framework as the main drive for the generated exercises, but the difference is how it uses the allowed list of exercises to force the models to create exercises that are only part of the framework.

This thesis extends the state of the art by demonstrating that integrating the more rigid cog-

nitive stage progression of the Neo-Piagetian framework via few-shot prompting effectively generates adequate exercises with the aim of helping students fully understand a programming topic and not only to generate practice exercises.

8.3 The *Predictive Engine*: Improvements in Knowledge Tracing

The third step of the tutoring system is the *Predictive Engine*, which aims to create a long-term analysis of the student-model interactions, in order to predict the risk of failing the final C programming exam. The prediction model was based on a Bi-LSTM with attention architecture that encapsulates all the interactions made by the student and the LLM in the remediation loop. This means that all the student's answers are part of the input for the predictive model captured as programming interactions over time, which represents the learning trajectory of each student. These interactions are composed of the enriched labels provided by the Neo-Piaget learning framework, which are: actual cognitive stage, missing skills, and mastered skills.
As detailed in section 6.3, the use of these enriched labels in the predictive model achieved an accuracy of 76% and an F1-score of the risk outcomes of 76%. These results represent a great improvement over the baseline knowledge tracing methodology, which uses the same proposed machine learning architecture, getting an accuracy of 67% and an F1-score value of 71%. These results were subject to the size of the training dataset, which was relatively small due to its dependence on only Moodle logs.
These final metrics answer the third research question, demonstrating that the use of cognitively rich labels significantly improves the prediction model. This is due to the fact that traditional knowledge tracing methods only assume the final outcome of an exercise to decide if the student knows the topic or not. Instead, the proposed methodology understands that a failure in one exercise does not mean a lack of knowledge, but it really depends on how well the student is evolving through the defined cognitive stages. By utilizing the fine-tuned LLM as an advanced feature extractor diagnosis, the system enriched the sequential data with the "why" behind the failure. In this way, the neural network is understanding the reasons behind a failure and not only the final grade. In addition, the use of attention maps during the training phase allows us to understand the importance of paying attention to the entire learning trajectory and not to the final interactions. It is because for the model it is relevant to detect the misconceptions a student could have during the course and not only during the last topics.

These results offer a distinction from the actual literature. For instance [20], uses LLMs conversations as input for knowledge tracing with deep neural networks but using only binary sequences. Although recent literature, such as [37] implements novel methods for generating semantic rich embeddings using LLMs, this approach bases its input only on general generated content, not a well-defined learning framework as the one used in this project. The proposed architecture bridges this gap by leveraging the semantic understanding of Large Language Models to extract structured pedagogical embeddings based on the Neo-Piagetian stages, which are then sequentially processed by a Bi-LSTM.
Finally, the improved results of the *Predictive Engine* close the entire proposed system, by creating a complete tutoring system that uses the Neo-Piaget learning framework to diagnose initial raw exercises, then generate tailored exercises that guide the learning trajectory, to finally use all those interactions for a final prediction of the risk of failure in a final programming exam.

8.4 Pilot Deployment Viability and Future Work Directions

The pilot deployment of the autonomous system successfully demonstrated the feasibility of integrating advanced LLM techniques with the cognitive stage definitions of the Neo-Piaget learning framework into a real educational environment. By establishing a stable learning loop between the student interface, the fine-tuned diagnostic model, and the exercise *Remediation Engine*, the system proved its viability. As presented in the results section, some study cases show that students can successfully follow the learning trajectory until they master the skills based on different cognitive stages. However, the behavioral patterns observed in some students who try to bypass the learning path revealed a fundamental conflict between the strict learning framework and user autonomy.
Because the primary objective of this pilot deployment was to validate all integration and system stability, data collection was done by saving logs in the database. Therefore, due to the voluntary nature of the tool's usage in a live classroom setting, formal qualitative evaluations of student perception, interface usability, and overall satisfaction yielded insufficient response rates. Although the interaction logs confirm that the models operate as intended, the lack of user perception surveys represents a limitation of the current evaluation.
The work on the perception gap will serve as the main focus of future work, in which the system is expected to transition from passive log interactions to active user engagement metrics. In

order to achieve the final user surveys, some aspects should be considered to encourage student interaction with the system and survey participation. Consequently, future studies should generate standardized usability questionnaires, such as evaluating system usability, model hallucinations from the student perspective, and general understanding of the learning flow.

9 Conclusions

Students in introductory C programming courses frequently struggle to understand complex topics because their learning demands a progressive understanding of abstract concepts rather than just syntax memorization. Furthermore, traditional artificial intelligence tools do not consider this hierarchical learning need; instead, they act primarily as code solvers. To overcome this problem, this project proposed and implemented an autonomous system that integrates Large Language Models with knowledge tracing under the constraints of the Neo-Piagetian learning framework. The developed solution creates a tool capable of diagnosing a student's cognitive state directly from raw C code submissions, generating tailored remedial exercises, and predicting academic risk failure in final examinations. These three components were consolidated into a functional web application, demonstrating the technical viability of the use of AI as a programming tutor based on a well-defined learning framework.

The evaluation of the three main components of the system yielded significant findings regarding its application in programming education. First, the development of the *Diagnostic Engine* established that a fine-tuned open-weights model, such as GPT-OSS-20b, can accurately extract cognitive patterns from raw student code and translate them into an established cognitive stage and specific skills to be practiced within the Neo-Piaget learning framework. This component was trained over historical Moodle logs data and achieved a 74% accuracy in the diagnostic task, which represents a significant improvement over the 66% achieved by the base model, while additionally outperforming general-purpose commercial models such as Gemini 2.5-Flash. This confirms that the fine tuning method can substantially increase the reasoning capabilities of an

open-weight model in order to perform a specific diagnostic task.

Second, the *Remediation Engine*, as the one responsible for generating tailored exercises based on the previous diagnosis, highlighted a critical limitation of LLMs: without explicit boundaries, LLMs cannot accurately follow the constraints of a learning framework in order to act as a programming tutor. This project demonstrated that applying few-shot prompting techniques with the use of a predefined typology of exercises successfully forced the models to adhere to the strict cognitive progression of the Neo-Piaget framework. This approach proved that LLMs are capable of generating complex and varied C programming exercises by explicitly defining the correct type of exercises in the prompt architecture.

Third, the next component of this learning loop is the *Predictive Engine*, which uses the history of student-model interactions to forecast the risk of failing a final exam. The training process focused on the use of a Bi-LSTM network as the architecture for this knowledge tracing model, in which the integration of the LLM generated labels for the input of the model significantly improved the predictive accuracy from a 67% baseline to 76%. This enriched label corresponds to the ones generated in the previous components: cognitive stage, missing skills, and mastered skills, which are based on the Neo-Piaget learning framework. This confirms that traditional knowledge tracing, which only tracks whether an answer is right or wrong, has limitations; for that reason, by supplying the predictive model with the complete reasoning behind a failure, the network achieved a highly accurate view of the student's trajectory.

Finally, the three components were consolidated in a web application that was deployed as a pilot in a real class environment, with the aim of validating the platform and model stability. The interactions between the real C programming students and the model proved that the system functions correctly in a classroom setting; however, the lack of qualitative feedback from the students represented a limitation in understanding students' perception. Consequently, future work should focus on multiple studies that allow the measurement of the real usability of the system, and more importantly how the continuous use of this tutoring system can improve the students' understanding of C programming topics and how it affects the outcome of the final examination.

In conclusion, the final proposed system is intended to help students understand C programming topics, based on progressive learning of abstract concepts following a learning framework; in which the use of specialized LLMs and knowledge tracing techniques create a complete learning tool that bridges the gap between automated code generation and genuine instructional guidance.

References

[1] C. Schulte, "Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching," in *Proceedings of the Fourth international Workshop on Computing Education Research*, ser. ICER '08. New York, NY, USA: Association for Computing Machinery, Sep. 2008, pp. 149–160. [Online]. Available: https://dl.acm.org/doi/10.1145/1404520.1404535

[2] J. Sorva, *Visual program simulation in introductory programming education*. Aalto University, 2012. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/3534

[3] R. Lister, "On the cognitive development of the novice programmer: and the development of a computing education researcher," in *Proceedings of the 9th Computer Science Education Research Conference*. Virtual Event Netherlands: ACM, Oct. 2020, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3442481.3442498

[4] W. Lyu, Y. Wang, T. R. Chung, Y. Sun, and Y. Zhang, "Evaluating the Effectiveness of LLMs in Introductory Computer Science Education: A Semester-Long Field Study," in *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. Atlanta GA USA: ACM, Jul. 2024, pp. 63–74. [Online]. Available: https://dl.acm.org/doi/10.1145/3657604.3662036

[5] Q. Fu, Y. Zhao, Z. Jia, and Y. Zheng, "Large Language Models (LLMs) in Programming Learning: The Current Research State and Agenda," *IEEE Transactions on Learning Technologies*, vol. 18, pp. 942–961, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11204697/

[6] R. E. Wang, A. T. Ribeiro, C. D. Robinson, S. Loeb, and D. Demszky, "Tutor CoPilot: A Human-AI Approach for Scaling Real-Time Expertise," Jan. 2025, arXiv:2410.03017 [cs]. [Online]. Available: http://arxiv.org/abs/2410.03017

[7] Z. Chu, S. Wang, J. Xie, T. Zhu, Y. Yan, J. Ye, A. Zhong, X. Hu, J. Liang, P. S. Yu, and Q. Wen, "LLM Agents for Education: Advances and Applications," Mar. 2025, arXiv:2503.11733 [cs]. [Online]. Available: http://arxiv.org/abs/2503.11733

[8] A. Kucheria, N. Sawhney, and A. Hellas, "Comparing behavioral patterns of llm and human tutors: A population-level analysis with the cima dataset," in *Workshop on Innovative Use of NLP for Building Educational Applications. Association for Computational Linguistics*, 2025.

[9] L. Krupp, S. Steinert, M. Kiefer-Emmanouilidis, K. E. Avila, P. Lukowicz, J. Kuhn, S. Küchemann, and J. Karolus, "Challenges and Opportunities of Moderating Usage of Large Language Models in Education," in *Proceedings of the 23rd International Conference on Mobile and Ubiquitous Multimedia*, ser. MUM '24. New York, NY, USA: Association for Computing Machinery, Dec. 2024, pp. 249–254. [Online]. Available: https://dl.acm.org/doi/10.1145/3701571.3701590

[10] A. T. Neumann, Y. Yin, S. Sowe, S. Decker, and M. Jarke, "An LLM-Driven Chatbot in Higher Education for Databases and Information Systems," *IEEE Transactions on Education*, vol. 68, no. 1, pp. 103–116, Feb. 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10706931

[11] E. Dickey, A. Bejarano, and C. Garg, "AI-Lab: A Framework for Introducing Generative Artificial Intelligence Tools in Computer Programming Courses," *SN Computer Science*, vol. 5, no. 6, p. 720, Jul. 2024. [Online]. Available: https://doi.org/10.1007/s42979-024-03074-y

[12] M. Mudongo, T. Taukobong, B. Gopolang, Y. Ayalew, E. Zimudzi, and E. Tshukudu, "Understanding Java Programming Errors by First Year Students through the Lens of the Neo-Piaget Cognitive Development Framework," in *Proceedings of the ACM Global Computing Education Conference 2025 - Volume 1*, ser. CompEd 2025, vol. 1. New York, NY, USA: Association for Computing Machinery, Oct. 2025, pp. 141–148. [Online]. Available: https://dl.acm.org/doi/10.1145/3736181.3754328

[13] R. Lister, *Concrete and other neo-piagetian forms of reasoning in the novice programmer*, Dec. 2011. [Online]. Available: https://opus.lib.uts.edu.au/handle/10453/17580

[14] R. Gluga, J. Kay, R. Lister, and D. Teague, "On the reliability of classifying programming tasks using a neo-piagetian theory of cognitive development," in

Proceedings of the ninth annual international conference on International computing education research, ser. ICER '12. New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 31–38. [Online]. Available: https://dl.acm.org/doi/10.1145/2361276.2361284

[15] S. Zhang, P. S. Meshram, P. Ganapathy Prasad, M. Israel, and S. Bhat, "An LLM-Based Framework for Simulating, Classifying, and Correcting Students' Programming Knowledge with the SOLO Taxonomy," in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2*, ser. SIGCSETS 2025. New York, NY, USA: Association for Computing Machinery, Feb. 2025, pp. 1681–1682. [Online]. Available: https://dl.acm.org/doi/10.1145/3641555.3705125

[16] M. Alidadi, F. Taghiyareh, and N. Shahhoseini, "Evaluating LLM-Generated Persian Questions for Teaching Conditional Programming Using Bloom's Taxonomy," in *2024 11th International Symposium on Telecommunications (IST)*, Oct. 2024, pp. 726–730. [Online]. Available: https://ieeexplore.ieee.org/document/10843532/

[17] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein, "Deep Knowledge Tracing," in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/bac9162b47c56fc8a4d2a519803d51b3-Abstract.html

[18] A. F. B. Amaya, "Academic performance prediction system based on machine learning models and knowledge tracing."

[19] L. Li, Z. Wang, J. M. Jose, and X. Ge, "LLM supporting knowledge tracing leveraging global subject and student specific knowledge graphs," *Information Fusion*, vol. 126, p. 103577, Feb. 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1566253525006499

[20] A. Scarlatos, R. S. Baker, and A. Lan, "Exploring Knowledge Tracing in Tutor-Student Dialogues using LLMs," in *Proceedings of the 15th International Learning Analytics and Knowledge Conference*. Dublin Ireland: ACM, Mar. 2025, pp. 249–259. [Online]. Available: https://dl.acm.org/doi/10.1145/3706468.3706501

[21] J. Piaget and B. Inhelder, *Psychology Of The Child*. Basic Books, 1969. [Online]. Available: https://books.google.com.ec/books?id=jtV-AAAAMAAJ

[22] C. Izu, C. Schulte, A. Aggarwal, Q. Cutts, R. Duran, M. Gutica, B. Heinemann, E. Kraemer, V. Lonati, C. Mirolo, and R. Weeda, "Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. Aberdeen Scotland Uk: ACM, Dec. 2019, pp. 27–52. [Online]. Available: https://dl.acm.org/doi/10.1145/3344429.3372501

[23] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A Comprehensive Overview of Large Language Models," *ACM Trans. Intell. Syst. Technol.*, vol. 16, no. 5, pp. 106:1–106:72, Aug. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3744746

[24] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt Engineering in Large Language Models," in *Data Intelligence and Cognitive Informatics*, I. J. Jacob, S. Piramuthu, and P. Falkowski-Gilski, Eds. Singapore: Springer Nature, 2024, pp. 387–402.

[25] "What is Fine-Tuning? | IBM." [Online]. Available: https://www.ibm.com/think/topics/fine-tuning

[26] A. Shirgaonkar, N. Pandey, N. C. Abay, T. Aktas, and V. Aski, "Knowledge Distillation Using Frontier Open-source LLMs: Generalizability and the Role of Synthetic Data," Oct. 2024, arXiv:2410.18588 [cs]. [Online]. Available: http://arxiv.org/abs/2410.18588

[27] J. Noble, "What is Long Short-term Memory (LSTM)? | IBM," Dec. 2025. [Online]. Available: https://www.ibm.com/think/topics/lstm

[28] R. Lister, "Toward a Developmental Epistemology of Computer Programming," in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. Münster Germany: ACM, Oct. 2016, pp. 5–16. [Online]. Available: https://dl.acm.org/doi/10.1145/2978249.2978251

[29] M. Lopez, J. Whalley, P. Robbins, and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," in *Proceedings of the Fourth international Workshop on Computing Education Research*. Sydney Australia: ACM, Sep. 2008, pp. 101–112. [Online]. Available: https://dl.acm.org/doi/10.1145/1404520.1404531

[30] Acm Computing Curricula Task Force, Ed., *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, Inc, Jan. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534860

[31] v. Team, "vLLM." [Online]. Available: https://vllm.ai

[32] X. Lu and X. Wang, "Generative Students: Using LLM-Simulated Student Profiles to Support Question Item Evaluation," in *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. Atlanta GA USA: ACM, Jul. 2024, pp. 16–27. [Online]. Available: https://dl.acm.org/doi/10.1145/3657604.3662031

[33] OpenAI, S. Agarwal, L. Ahmad, J. Ai, S. Altman, A. Applebaum, E. Arbus, R. K. Arora, Y. Bai, B. Baker, H. Bao, B. Barak, A. Bennett, T. Bertao, N. Brett, E. Brevdo, G. Brockman, S. Bubeck, C. Chang, K. Chen, M. Chen, E. Cheung, A. Clark, D. Cook, M. Dukhan, C. Dvorak, K. Fives, V. Fomenko, T. Garipov, K. Georgiev, M. Glaese, T. Gogineni, A. Goucher, L. Gross, K. G. Guzman, J. Hallman, J. Hehir, J. Heidecke, A. Helyar, H. Hu, R. Huet, J. Huh, S. Jain, Z. Johnson, C. Koch, I. Kofman, D. Kundel, J. Kwon, V. Kyrylov, E. Y. Le, G. Leclerc, J. P. Lennon, S. Lessans, M. Lezcano-Casado, Y. Li, Z. Li, J. Lin, J. Liss, Lily, Liu, J. Liu, K. Lu, C. Lu, Z. Martinovic, L. McCallum, J. McGrath, S. McKinney, A. McLaughlin, S. Mei, S. Mostovoy, T. Mu, G. Myles, A. Neitz, A. Nichol, J. Pachocki, A. Paino, D. Palmie, A. Pantuliano, G. Parascandolo, J. Park, L. Pathak, C. Paz, L. Peran, D. Pimenov, M. Pokrass, E. Proehl, H. Qiu, G. Raila, F. Raso, H. Ren, K. Richardson, D. Robinson, B. Rotsted, H. Salman, S. Sanjeev, M. Schwarzer, D. Sculley, H. Sikchi, K. Simon, K. Singhal, Y. Song, D. Stuckey, Z. Sun, P. Tillet, S. Toizer, F. Tsimpourlas, N. Vyas, E. Wallace, X. Wang, M. Wang, O. Watkins, K. Weil, A. Wendling, K. Whinnery, C. Whitney, H. Wong, L. Yang, Y. Yang, M. Yasunaga, K. Ying, W. Zaremba, W. Zhan, C. Zhang, B. Zhang, E. Zhang, and S. Zhao, "gpt-oss-120b & gpt-oss-20b Model Card," Aug. 2025, arXiv:2508.10925 [cs]. [Online]. Available: http://arxiv.org/abs/2508.10925

[34] N. B. D. Ta, H. G. P. Nguyen, and S. Gottipati, "ExGen: Ready-To-Use Exercise Generation in Introductory Programming Courses," *International Conference on Computers in Education*, Dec. 2023. [Online]. Available: https://library.apsce.net/index.php/ICCE/article/view/953

[35] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ser. ICER '22, vol. 1. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 27–43. [Online]. Available: https://dl.acm.org/doi/10.1145/3501385.3543957

[36] H. Wu, D. Fa, X. Wu, W. Tan, X. Chang, Y. Gao, and J. Weng, "Research on the Construction of Intelligent Programming Platform Based on AI-generated Content," in *Proceedings of the 15th International Conference on Education Technology and Computers*, ser. ICETC '23. New York, NY, USA: Association for Computing Machinery, Jan. 2024, pp. 9–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3629296.3629298

[37] Y. Ozyurt, S. Feuerriegel, and M. Sachan, "Automated Knowledge Concept Annotation and Question Representation Learning for Knowledge Tracing," Mar. 2025, arXiv:2410.01727 [cs]. [Online]. Available: http://arxiv.org/abs/2410.01727