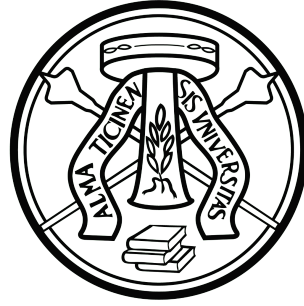




University of Pavia
Faculty of Engineering
Department of Electrical, Computer and
Biomedical Engineering

Master Degree in Intelligent Control System

CheckMyC - Qualitative Code Evaluator Using Large Language Models

Supervisor:
Prof. Tullio Facchinetti

Candidate:
Federico Romano

Academic Year 2024/2025

Abstract

Traditional automated assessment of programming assignments relies on functional testing based on predefined inputs and expected outputs. While effective for detecting functional defects, this approach fails to measure qualitative aspects central to educational goals.

This thesis investigates the reliability of Large Language Models (LLMs) when used to support qualitative assessment of student-written source code in academic contexts. The study is guided by three research questions: (RQ1) the degree of correspondence between LLMs and human evaluations; (RQ2) the stability and internal reliability of evaluations across repeated applications; and (RQ3) the systematic differences and comparative reliability among multiple LLMs under identical criteria.

To address these questions, the thesis proposes CheckMyC, an evaluation framework designed to produce structured, traceable, and comparable qualitative assessments. The framework combines a topic-based rubric with constrained prompting strategies and rigid output schemas, enabling LLMs to generate localized evidences and per-topic scores while avoiding unstructured or purely subjective judgments. The methodology is implemented through a modular pipeline that performs repeated LLM-based evaluations across multiple models and programs, followed by post-processing and statistical analysis on a dataset of 44 C programs.

The experimental results on this dataset show that high-capacity models like Gemini 2.5 Pro achieve superior alignment with human judgment, reaching high detection ability for most of the defined topics conditions, and high stability in the responses. In contrast, smaller architectures exhibit significant limitations due to omission bias and high hallucination rates.

The study concludes that model architecture is the primary determinant of performance. While large models demonstrate the reliability necessary for student self-assessment and instructional support, smaller models remain unsuitable for academic grading due to stochastic noise and intrinsic consistency limitations.

Contents

Contents	v
List of Figures	ix
List of Tables	xi
1 Introduction	3
1.1 Problem Statement	4
1.2 Objective and Research Questions	5
1.3 Contributions	6
1.4 Thesis Structure	7
2 State of the art	9
2.1 Automated Code Assessment in Academic Contexts	9
2.2 Traditional Automatic Grading Systems	11
2.3 Static and Dynamic Code Analysis	13
2.4 LLMs for Automatic Code Grading	16
2.5 Reliability and Consistency of LLM-Based Evaluation	19
2.6 Hallucinations and Bias in LLM-Based Assessment	21
2.7 Evaluation of the Generated Explanations Quality	24
2.8 Critical Synthesis of the State of the Art	26
3 Tools and Frameworks	29
3.1 Large Language Models and Inference Services	29
3.1.1 Generative AI Models	30
3.1.2 Application Programming Interfaces (API)	31
3.1.3 Software Development Kits (SDK)	32
3.2 Compilers and Functional Testing Tools	32
3.2.1 The GNU Compiler Collection (GCC)	33

3.2.2	pvcheck Testing Framework	34
3.3	Programming Environment and Software Libraries	34
3.3.1	The Python Programming Language	35
3.3.2	Templating and Configuration Tools	36
3.3.3	Data Analysis and Statistical Libraries	36
3.3.4	Visualization and Reporting Frameworks	37
4	Implementation	39
4.1	Design Goals of the Framework	39
4.2	Logical Architecture	41
4.3	System Model and Formal Notation	43
4.3.1	Evaluation Entities	43
4.3.2	Weighting Schemes: Percentage vs. Fixed	45
4.4	Execution Pipeline and Workflow	45
4.5	Configuration Mechanisms	48
4.5.1	System Orchestration	48
4.5.2	Pedagogical Rules and LLM Scoring	49
4.5.3	Objective Testing and Requirements	49
4.5.4	Course-Level Adaptability	49
4.6	Prompt Engineering and Structured Output	50
4.6.1	System and User Prompts	50
4.6.2	Topic-Specific Prompts	54
4.6.3	Structured Output Schema	56
4.7	Multi-Provider Abstraction Layer	57
4.7.1	The Provider Pattern Strategy	57
4.7.2	Robustness and Error Handling	58
4.7.3	Experimental Control and Determinism	58
4.8	Evaluation Models and Output Generation	60
4.8.1	Level 1: Atomic Evaluations (Per-Topic and Per-Test)	60
4.8.2	Level 2: Qualitative and Functional Aggregation	61
4.8.3	Level 3: Final Grade Synthesis (S_{Final})	62
4.8.4	Artifact Materialization: JSON and HTML	62
4.9	Analysis Component	63
4.9.1	Experimental Dataset and Human Gold Standard	64
4.9.2	Weight Calibration and Per-Topic Optimization	65
4.9.3	Metrics for Scoring and Detection Accuracy	68

4.9.4	Stability and Reliability Analysis	71
5	Results	73
5.1	Human-LLM Alignment Analysis	74
5.1.1	LLM Evidence Extraction Capabilities	74
5.1.2	LLM Score Alignment Analysis	78
5.2	Stability and Reliability Results	83
5.2.1	Qualitative Evidence Stability	83
5.2.2	Quantitative Score Reliability	84
5.2.3	Summary of Stability Findings	85
6	Conclusions	87
6.1	Future Work	88
	Bibliography	89

List of Figures

2.1	Xiang et al. automatic assessment technique based on program dependency graph similarity.	14
2.2	Consistency heatmaps of Messer et al. study between different evaluators.	21
2.3	Average hallucination rate per prompt type.	22
2.4	Self-consistency method example with multiple runs	24
4.1	High-level overview of the design goals and users.	41
4.2	Logical architecture of the framework, showing the main layers and their interactions.	42
4.3	Frequency of defined condition types (excluding neutral) across the assessment topics.	55
4.4	User-friendly visual report of the <i>checkmyc</i> output obtained by the	63
4.5	Optimization results (Comparison and Convergence) for the first three topics.	69
4.6	Optimization results for Error Handling and Code Quality.	70
5.1	Precision-Recall scatter plots of Gemini models.	75
5.2	Precision-Recall scatter plots of GPT models.	76
5.3	Hallucination rate of LLMs across topics	79
5.4	MAE Gap for reliable models across topics.	81
5.5	Bias Gap for reliable models across topics.	81
5.6	Spearman Coefficient Gap for reliable models across topics.	82
5.7	Jaccard index heatmap for models across all topics.	84
5.8	Intraclass Correlation Coefficient (ICC(1,k)) heatmap for models across all topics.	85
5.9	Standard Deviation (STD) heatmap for models across all topics. . .	86

List of Tables

3.1	Comparative overview of the LLMs utilized in the framework.	31
4.1	Notation and valid domains for condition subscripts.	44
4.2	Validity of Goodness-Criticality pairs.	44
4.3	MAE computed on the full dataset ($N = 44$) using Leave-One-Out Cross-Validation (LOOCV) to estimate generalization performance. .	67
4.4	Final Optimized Metrics for each topic after calibration on the full dataset ($N = 44$)	68
5.1	Average Evidence Extraction Metrics per Model.	78
5.2	Detailed Scoring Metrics and Tech Gaps per Topic and Model. . . .	80
5.3	Interpretation of Technological Gaps for Scoring Metrics.	80

Acronyms

CoT Chain of Thought

LLM Large Language Model

AST Abstract Syntax Trees

API Application Programming Interface

SDK Software Development Kit

LOOCV Leave-One-Out Cross-Validation

MAE Mean Absolute Error

ICC(1,k) Intraclass Correlation Coefficient

STD Standard Deviation

P Precision

R Recall

Chapter 1

Introduction

The increasing adoption of Large Language Model (LLM) has led to their widespread utilization as auxiliary tools in software development workflows. Their use is not limited to code generation but includes different activities such as code inspection, explanation, debugging support and qualitative analysis. As these tools have become increasingly accessible by everyone, their potential application in academic environments has attracted growing attention, particularly in programming courses where the evaluation of student work represents a recurring and resource-intensive task.

In these courses, assessment typically relies on automated testing leading to an evaluation that mainly takes into account the functionality of the code, leaving aside its quality. This evaluation method is used for its determinism but could risk invalidating valuable students' exams that, due to small errors, fail the automated testing. For this reason manual inspection of student-written source code is crucial to account for many aspects that are central to instructional goals. Consequently, qualitative evaluation remains an essential component of the assessment process, despite its inherent cost and limited scalability.

Manual evaluation is further affected by structural limitations. Human judgments may vary due to subjective interpretation, accumulated fatigue, or constraints such as class size and time. Even when detailed grading guidelines are available, qualitative assessments can differ across evaluators or even across different evaluation sessions. These inconsistencies become increasingly problematic in large academic contexts, where ensuring uniformity and fairness across a substantial number of submissions is particularly challenging.

For these limitations, LLMs can be considered as potential useful decision-support systems rather than replacements for human evaluators. Their simultaneous capacity to interpret natural language criteria, analyze code and generate structured

qualitative feedback suggests that they may assist instructors by offering preliminary evaluations or highlighting relevant aspects of a submission, as well as students by offering an auto-evaluator tool that is aligned with a particular course program. If their behaviour shows sufficient reliability, such systems could help instructors by lightening evaluative workload while supporting greater consistency in qualitative assessment, and students by providing them a personal course-specific self-evaluator.

1.1 Problem Statement

Assessing source code written by students within academic environments is a task that is fundamentally qualitative and strongly influenced by context. Unlike problems that can be evaluated solely through formal specifications or objective correctness criteria, educational code assessment needs to consider aspects such as the student's understanding of core concepts, the suitability of design decisions, the program's behaviour with invalid inputs, and the correct implementation of the course topics. These factors go beyond what can be reliably measured through automated test cases or rule-based analyses, and instead depend on interpretative judgment shaped by educational objectives.

A central difficulty in this domain is represented by the lack of a formal and deterministic ground truth that can be directly verified by automated means. For any given program, there is no single, definitive qualitative evaluation that can serve as an objective benchmark. Evaluations are instead produced by experienced instructors or teaching assistants and therefore act as human reference judgments, rather than as absolute or formally specified machine-verifiable ground truth. In fact, these judgments are informed by pedagogical goals, prior experience, and contextual expectations, that can be more relaxed in certain topics with respect to others. Although these evaluations carry authority within the academic setting, they cannot be translated into a fixed specification that is directly verifiable by a machine.

This absence of a formal, machine-verifiable ground truth also complicates the comparison of assessments. Differences may emerge not only between automated systems and human evaluators, but also among human evaluators themselves. In practice, evaluations are compared indirectly, often through alignment with grading rubrics, reference implementations, or exemplar solutions. Even under standardized conditions, qualitative variation does not necessarily imply incorrectness; rather, it may reflect legitimate alternative interpretations of the same code.

The introduction of LLMs into this scenario further intensifies these challenges.

LLMs produce evaluations that depend on various aspects: prompt structure, contextual information, and inherent stochastic mechanisms. For this reason, their outputs cannot be assumed to be identical across multiple executions, nor can they be straightforwardly measured against a single reference answer. Evaluation strategies that rely on exact matching or predefined outputs are therefore ill-suited to this setting.

This thesis addresses the assessment of the reliability of qualitative code evaluations generated by LLMs in situations where no formal, machine-verifiable ground truth exists and evaluation relies instead on human reference judgments. In this context, reliability is not to be considered as a factual correctness, but as the extent to which the generated evaluations are coherent, consistent, and meaningful aligned with human reference judgments in an educational framework. To tackle this issue, moving away from binary evaluation schemes is necessary together with examining relational and structural characteristics of the assessments.

Accordingly, this work focuses on defining and applying criteria that enable the systematic comparison of qualitative evaluations produced by LLMs, both in relation to human reference assessments and across different models, under conditions representative of real academic practice.

1.2 Objective and Research Questions

The aim of this thesis is to examine the extent to which LLMs can be considered reliable when used to assist in the qualitative assessment of source code in academic environments. Rather than considering these models as independent evaluators, the study approaches them as support tools whose value is determined by the degree to which their assessments remain interpretable, coherent, and compatible with judgments provided by human evaluators. From this perspective, reliability is investigated along two distinct but related aspects. On one hand, the analysis considers how closely evaluations produced by LLMs correspond to reference assessments formulated by the instructor, which serve as the practical standard in educational context. On the other hand, the study examines the intra and inter behaviour of the models, with particular attention to their consistency and coherency.

Based on these considerations, the work is structured around the following research questions:

- **RQ1:** To what degree do assessments generated by Large Language Models correspond to human reference evaluations in an academic context?

- **RQ2:** How stable are the evaluations produced by a given LLM when applied to the same programs or evaluator dimensions, and how can this stability be interpreted in terms of internal reliability?
- **RQ3:** When multiple LLMs are applied to the same code under identical qualitative criteria, what systematic differences can be observed, and how do these differences influence their comparative reliability?

These research questions deliberately avoid assumptions of absolute correctness or the existence of a formal and deterministic ground truth. Instead, they focus on relational properties such as alignment with human judgment, consistency across evaluations, and variability among models, as concrete indicators through which the reliability of LLM-based support for code evaluation can be rigorously examined.

1.3 Contributions

This work contributes to the study of LLMs by proposing a structured and reproducible approach to examining the reliability in the qualitative evaluation of source code within academic settings. In particular, the *Fondamenti di Informatica* programming course delivered at the Department of Industrial, Computer and Biomedical Engineering of the University of Pavia has been considered as case study, in order to evaluate the LLMs capacity in qualitative assessments of students exam programs under prompted instructor guidelines.

A primary contribution is the design of an evaluation framework specifically intended to examine assessments generated by LLMs under controlled experimental conditions. The framework is conceived to obtain qualitative judgments, rather than binary scores of correctness, and support the systematic collection and comparison of model outputs. For this reason, particular attention is assigned to the adoption of rigid output structures, which enable consistent interpretation, facilitate automated analysis, and support traceable post-processing procedures.

In addition, the thesis introduces a methodological perspective for analyzing reliability in contexts where no formal, machine-verifiable truth is available. Human evaluations are treated as reference judgments instead of absolute benchmarks, allowing reliability to be investigated through observable characteristics such as consistency with reference assessments, coherency across related evaluations, and divergence among different models. This perspective makes it possible to reason about reliability without assuming that qualitative assessment can be fully formalized or reduced to deterministic rules.

The work further contributes an annotated collection of C programs paired with qualitative evaluations, constructed to reflect realistic instructional scenarios. The dataset is designed to support comparative analyses across both evaluation topics and language models, and to provide a stable empirical basis for identifying recurring patterns in LLM-generated feedback.

Finally, the thesis offers a critical examination of the outcomes produced by the proposed framework. This analysis addresses both the strengths and the limitations of the used LLMs when employed as support tools for academic code evaluation, clarifying the circumstances under which they may offer practical benefits and the constraints that must be considered when integrating them into educational assessment processes.

1.4 Thesis Structure

The thesis is structured into six chapters, each addressing a specific aspect of the study.

Chapter 1 establishes the background of the work, introduces the central problem under examination, and defines the objectives and research questions that guide the analysis. It also outlines the main contributions of the thesis.

Chapter 2 provides an overview of the existing literature relevant to this work. It discusses prior approaches to automatic code evaluation, examines the use of LLMs for code analysis, and reviews current methods for assessing LLM-generated outputs, with particular emphasis on their limitations in qualitative and educational settings.

Chapter 3 describes the tools and methodological components adopted in the study. This includes an overview of the language models considered, the strategies employed for prompt formulation and output structuring, and the techniques used for representing and clustering textual evaluations.

Chapter 4 details the implementation of the proposed framework. It explains how the dataset was constructed, how prompts were designed, how model evaluations were executed, and how the resulting outputs were processed and aggregated for analysis.

Chapter 5 presents the results obtained from the experimental evaluation. It includes quantitative analyses, examines patterns emerging from semantic clustering, and compares the behavior of different models, followed by an interpretation of the observed findings.

Chapter 6 concludes the thesis by reflecting on the main outcomes of the study, discussing its limitations, and indicating possible directions for future research.

Chapter 2

State of the art

This chapter presents an overview of the current state of research relevant to the objectives of this thesis. It introduces the theoretical foundations, methodological approaches, and technological frameworks that underpin the proposed work. The discussion begins with the core concepts necessary to contextualize the problem domain, followed by an analysis of existing solutions and evaluation methodologies. Particular attention is given to approaches that leverage LLMs for code analysis and quality assessment, highlighting their strengths, limitations, and open challenges. This structured overview establishes the conceptual and technical background required to motivate and justify the contributions developed in the subsequent chapters.

2.1 Automated Code Assessment in Academic Contexts

The assessment of source code is a central component of computer science education, particularly in introductory and intermediate programming courses. Its primary objective is not only to verify whether a program produces the expected output, but also to evaluate the student’s understanding of fundamental concepts, problem-solving strategies, and programming practices. In academic contexts, code assessment therefore plays a dual role: it serves as a mechanism for grading student performance and as a pedagogical tool to support learning through feedback.

Traditionally, programming assessment has been performed manually by instructors or teaching assistants, who inspect code submissions, execute them on selected inputs, and assign grades based on correctness and perceived quality. However, the rapid growth in student enrollments in computer science programs has made manual grading increasingly impractical. Large-scale courses, such as introductory programming classes, may involve hundreds or thousands of students, making timely

and consistent evaluation a significant challenge. This scalability issue has been one of the primary drivers behind the adoption of automated code assessment tools in academic environments [12, 7].

Beyond scalability, automated assessment is often motivated by the need for consistency and objectivity. Human grading is inherently subject to variability: different graders may apply evaluation criteria differently, and even the same grader may be inconsistent across time [19]. Automated systems, when properly configured, can apply the same criteria uniformly across all submissions, reducing unintended discrepancies. Nevertheless, consistency alone does not guarantee pedagogical effectiveness, as the nature of the criteria being applied remains crucial.

In educational settings, assessment has always been divided into formative and summative assessment. Formative assessment aims to support learning by providing feedback that helps students identify mistakes and improve their understanding, while summative assessment focuses on assigning a final grade that reflects the level of achievement at the end of an instructional unit [6]. In programming education, formative assessment is particularly important, as students often learn through iterative trial-and-error, progressively refining their solutions. Automated assessment systems are frequently used in formative contexts, such as online judges and practice platforms, where students can submit code multiple times and receive immediate feedback. Summative assessment, on the other hand, typically occurs in exams or graded assignments, where the goal is to measure student performance in a controlled and fair manner. In this context, automated code assessment is often used to support or complement traditional grading, for example by verifying functional correctness or detecting plagiarism. However, relying solely on automated criteria for summative evaluation raises concerns, particularly when the assessment reduces complex student work to a binary notion of correctness.

A key limitation of many automated assessment approaches is their tendency to focus on purely functional criteria. Functional correctness evaluates whether a program produces the correct output for a given set of inputs, often defined by a test suite. While correctness is a necessary condition for a valid solution, it is not sufficient to capture the broader educational objectives of programming courses. Aspects such as code readability, modularity, robustness to invalid inputs, and adherence to good programming practices are equally important in an academic context. These qualities are inherently qualitative and context-dependent, making them difficult to assess using rigid, predefined rules.

Consequently, code assessment in academia should be understood as a qualitative

and contextual process rather than a purely mechanical one. The interpretation of a student's solution depends on factors such as the learning objectives of the course, the expected level of expertise, and the specific constraints imposed by the assignment. For example, a solution that is acceptable in an introductory course may be considered inadequate in an advanced course due to inefficiency or poor design. Automated systems that ignore this context risk providing misleading evaluations, particularly when used as the sole basis for grading.

This tension between functional criteria and formative educational goals underpins much of the research on automated code assessment. While automation offers clear advantages in terms of scalability and efficiency, it also introduces the risk of oversimplifying evaluation to what can be easily measured. As a result, contemporary research increasingly emphasizes the need for assessment approaches that go beyond output correctness and attempt to capture higher-level properties of student code, as well as the reasoning behind evaluation decisions.

The limitations of traditional assessment practices and the growing demand for scalable yet pedagogically meaningful evaluation provide the foundation for the subsequent sections of this chapter. In particular, they motivate an examination of traditional automatic grading systems, their structural constraints, and the emergence of more advanced approaches, such as those based on LLMs, that aim to address the qualitative aspects of code assessment.

2.2 Traditional Automatic Grading Systems

Traditional automatic grading systems represent the earliest and most widely adopted approach to automated code assessment in academic settings. Their design is largely driven by simplicity, scalability, and determinism, with the primary goal of evaluating whether a student program satisfies a predefined specification. In most cases, these systems are built around the notion of functional correctness, operationalized through the execution of student code on a fixed set of test cases.

The most common paradigm is test-case based grading, where each submission is compiled and executed against an instructor-defined test suite. The program is considered correct if its output matches the expected output for all or a subset of the test cases. This approach is conceptually straightforward and easy to implement, making it particularly attractive for large-scale introductory courses. Test cases may range from simple input-output pairs to more complex unit tests that verify individual functions or modules, but the underlying principle remains the same:

correctness is inferred from observable behavior.

A closely related approach is output-based grading, in which the program's output is compared directly to a reference output, often using textual comparison tools. Minor variations in formatting may be tolerated through normalization or custom comparison scripts, but the evaluation remains fundamentally syntactic. These systems are widely used in online judges and exam environments due to their efficiency and ease of automation. However, they implicitly assume that the observable output fully captures the correctness and quality of a solution, an assumption that rarely holds in realistic educational scenarios.

One of the main strengths of traditional automatic grading systems lies in their scalability. Once the test infrastructure is in place, thousands of submissions can be evaluated with minimal additional cost, making these systems indispensable in mass education contexts [5]. Furthermore, because grading decisions are based on deterministic rules, such systems offer a high degree of objectivity: the same submission will always receive the same score, independent of external factors such as grader fatigue or subjective interpretation. This property is often cited as a major advantage over manual grading, particularly in summative assessment.

Despite these advantages, traditional automatic graders suffer from well documented structural limitations. Most notably, they provide little to no meaningful feedback beyond indicating which test cases failed. Students are typically informed that their solution is incorrect, but not why it is incorrect or how it could be improved. As a result, these systems are of limited pedagogical value, especially in formative settings where feedback is essential for learning [17]. The absence of explanatory feedback often forces students to rely on trial-and-error strategies rather than conceptual understanding.

Another significant limitation is fragility. Test-based graders are highly sensitive to deviations from the expected program structure, input format, or output formatting. A solution that is logically correct but differs slightly in presentation may be penalized or even receive a zero score. This fragility is particularly problematic in exam settings, where a single syntactic error can invalidate an otherwise correct solution. Unlike human graders, traditional systems are generally unable to assign partial credit or to recognize near-correct solutions.

Moreover, these systems tend to produce disproportionate penalties. Because grading is often binary at the level of individual test cases, small mistakes can lead to large reductions in score. For example, failing a single critical test may result in a zero grade, even if the rest of the solution demonstrates substantial

understanding. This behavior contrasts sharply with human grading practices, where partial correctness and reasoning are typically taken into account. As a consequence, students may be penalized more severely than intended, raising concerns about fairness and alignment with educational goals. Empirical studies show that test suites defined by instructors often suffer from insufficient behavioral coverage, leading to evaluations in which defective solutions are incorrectly classified as correct (false negatives in defect detection, false positives in grading). Clegg highlights how the application of Mutation Testing can reveal these structural deficiencies [4].

Traditional automatic grading systems also struggle with assignments that admit multiple valid solutions or require creative problem-solving. Since correctness is defined implicitly by the test suite, any solution that satisfies the tests is considered acceptable, regardless of its design quality, efficiency, or adherence to best practices. Conversely, unconventional but valid solutions may fail hidden assumptions embedded in the tests. This limitation forces instructors to constrain assignment design, often favoring narrowly specified problems over open-ended tasks. Recent empirical studies confirm that programming assignments allowing multiple valid solutions are extremely difficult to assess reliably using unit test-based approaches alone, as correctness cannot be exhaustively captured through predefined input-output pairs [23].

In summary, traditional automatic grading systems excel at enforcing functional specifications at scale, but they do so at the cost of pedagogical richness and flexibility. Their reliance on test cases and output comparison makes them effective tools for verifying correctness, yet poorly suited for assessing higher-level qualities of code or supporting meaningful feedback. These limitations motivate the exploration of complementary approaches, including static and dynamic code analysis techniques, which are discussed in the following section.

2.3 Static and Dynamic Code Analysis

To mitigate the limitations of purely output-based grading systems, research and practice have increasingly explored static and dynamic code analysis techniques as complementary tools for automated assessment. These approaches aim to inspect properties of source code beyond functional correctness, with the goal of capturing aspects related to structure, style, and behavior that are relevant in educational contexts.

Static code analysis examines program source code without executing it. In the context of academic assessment, static analysis is typically used to extract metrics

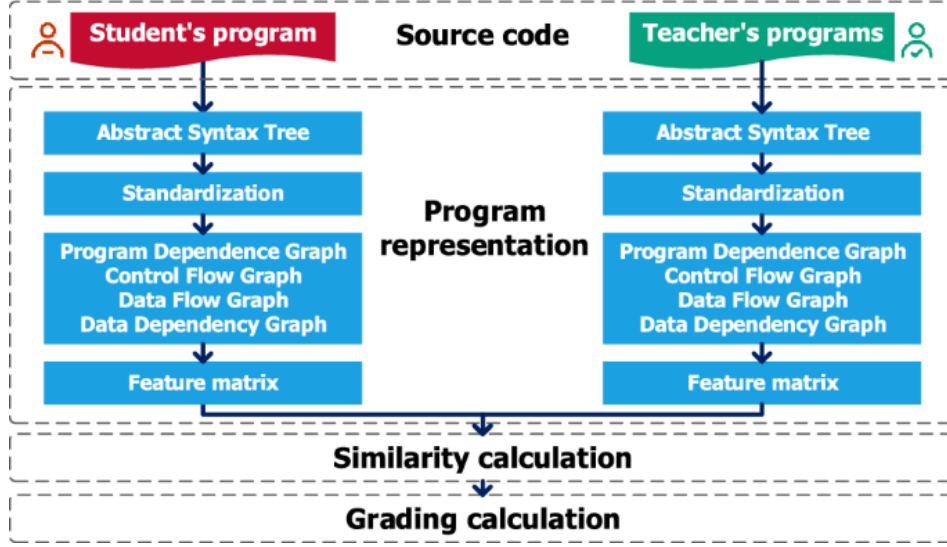


Figure 2.1: Algorithm flow of automatic assessment of program work based on the similarity of graph semantics. Source: [27].

and properties such as cyclomatic complexity, code length, nesting depth, use of control structures, naming conventions, and adherence to style guidelines. Tools originally designed for software quality assurance, such as linters and static analyzers, have been repurposed to provide automated feedback to students, often highlighting stylistic issues or potential defects. These metrics are attractive because they are inexpensive to compute, deterministic, and easily scalable to large cohorts.

More advanced static analysis techniques leverage Abstract Syntax Trees (ASTs), control-flow graphs, or dependency graphs to compare student solutions against reference implementations or to detect common misconceptions [17, 27]. Xiang et al. propose an automatic assessment technique based on program dependency graph similarity (see Figure 2.1), capable of assigning meaningful scores without executing the code, thereby enabling evaluation even in the presence of runtime errors or incomplete implementations [27]. In educational research, AST-based representations are frequently used as features for downstream analysis, including clustering of student solutions or machine learning-based grading. Static representations also enable plagiarism detection and similarity analysis, which are often integrated into grading pipelines to ensure academic integrity.

Despite their apparent usefulness, static metrics exhibit well-known limitations when applied to assessment. Many metrics correlate weakly with code quality or learning outcomes, particularly in novice-written code [5, 19]. A program with low cyclomatic complexity is not necessarily conceptually correct, and stylistic violations

may reflect lack of familiarity with conventions rather than misunderstanding of core concepts. Consequently, static analysis tends to favor surface-level properties and risks conflating syntactic regularity with conceptual understanding.

Dynamic code analysis, by contrast, evaluates program behavior at runtime. This category includes traditional test-case execution, but also more advanced techniques such as coverage analysis, mutation testing, and runtime tracing. In educational settings, dynamic analysis is commonly extended beyond simple input-output checking to assess how thoroughly a student solution exercises required functionality. For instance, code coverage metrics can indicate whether key branches or functions are reached during execution, providing a finer-grained notion of correctness than pass/fail test results.

Mutation testing has also been explored as a means of evaluating the robustness of student programs. By introducing small changes (mutants) into the code and observing whether test cases detect them, instructors can assess the adequacy of both the student's solution and the test suite itself [16]. While promising in theory, such techniques are computationally expensive and difficult to deploy at scale in typical course infrastructures, limiting their practical adoption.

Dynamic analysis can provide richer behavioral insights than static methods, but it remains fundamentally constrained by the quality and completeness of the test scenarios. As with traditional automatic graders, any behavior not exercised by the tests remains invisible to the system. Moreover, dynamic analysis offers limited support for evaluating non-functional aspects such as readability, abstraction quality, or algorithmic elegance, which are often central learning objectives in programming courses.

Integrating static and dynamic analyses into educational workflows introduces additional challenges. From an instructional perspective, the interpretation of metrics is non-trivial: students may receive warnings or scores without understanding their relevance to learning objectives. From a system design perspective, combining multiple analyses increases complexity and maintenance costs, particularly when assignments evolve or differ substantially across courses. As a result, many instructors selectively adopt only a subset of available analyses, often prioritizing those that are easy to explain and justify to students.

A more fundamental issue concerns alignment with high-level educational goals. While static and dynamic analyses can detect deviations from expected patterns or behaviors, they struggle to assess whether a student has chosen an appropriate problem-solving strategy or demonstrated meaningful understanding. For example,

two solutions with identical dynamic behavior may differ substantially in terms of decomposition, abstraction, or generality, yet such distinctions are difficult to capture with predefined metrics [17, 27]. This misalignment becomes increasingly pronounced as assignments grow more open-ended and emphasize design rather than mere correctness.

In summary, static and dynamic code analyses represent an important step beyond naive output-based grading, enabling the automated extraction of additional information about student programs. However, their reliance on predefined metrics and test scenarios limits their ability to capture the qualitative and contextual dimensions of programming competence. These shortcomings have motivated recent research into more flexible, semantics-aware approaches to assessment, setting the stage for the use of LLMs capable of reasoning about code structure, intent, and explanation, as discussed in the following section.

2.4 LLMs for Automatic Code Grading

The recent emergence of LLMs has introduced a qualitatively different approach to automated code assessment. Unlike traditional systems based on explicit rules, static metrics, or predefined test cases, LLMs operate by learning statistical regularities from large corpora of natural language and source code. This enables them to process code as a semantic artifact rather than as a purely syntactic object, opening new possibilities for grading, feedback generation, and qualitative evaluation.

Early applications of LLMs to programming tasks relied on models trained specifically on code, such as CodeBERT, GraphCodeBERT, and Codex. These code-specialized models are typically pretrained on large collections of source code repositories and associated documentation, allowing them to capture language-specific patterns, programming idioms, and structural regularities [9, 3]. In the context of automatic grading, such models have been used to classify student submissions, predict correctness, or detect common error patterns. Their main advantage lies in their strong alignment with programming languages: they are less prone to syntactic hallucinations and often perform well on tasks tightly coupled to code structure, such as defect prediction or similarity detection.

However, code-specialized models also exhibit important limitations when applied to academic grading. Most notably, they tend to operate at the level of prediction rather than explanation. While they can output labels or scores with reasonable accuracy, they are generally less effective at producing rich, pedagogically meaningful

feedback. Furthermore, many of these models require task-specific fine-tuning on labeled datasets, which are costly to construct and often tied to a particular course or assignment, limiting generalization and reuse [20].

More recently, general-purpose LLMs, such as GPT-3.5, GPT-4, and comparable transformer-based systems, have demonstrated strong performance on a wide range of code-related tasks without explicit fine-tuning. These models are trained on heterogeneous mixtures of natural language and code, enabling them to bridge the gap between formal program semantics and informal human explanations. As a result, they can not only assess whether code is correct, but also articulate why a solution is correct or incorrect, suggest improvements, and relate implementation details to high-level problem descriptions [3, 21].

Recent empirical studies support this claim. Fang et al. show that modern LLMs are able to perform non-trivial code analysis tasks, including bug localization and semantic reasoning over program structure, while also generating fluent natural language explanations of their judgments [8]. Similarly, Xie et al. demonstrate that general-purpose LLMs, when guided by rubric-aware prompting, can produce grading decisions and explanations that closely resemble those of human instructors in academic programming assessment contexts [28].

Beyond general-purpose and code-specialized pretrained models, recent work has explored the fine-tuning of large language models explicitly for grading tasks. Yousef et al. introduce BeGrading, a large language model fine-tuned on programming assignments and instructor feedback, demonstrating improved alignment with human grading and reduced evaluation effort in academic settings [29]. While such approaches can enhance grading accuracy, they require curated datasets and task-specific training, which may limit their applicability across different courses and institutions.

In the context of automated grading, general-purpose LLMs are typically employed through prompt-based evaluation. The grading criteria, assignment description, reference solution, and student submission are embedded into a structured prompt, and the model is instructed to produce an evaluation according to a pre-defined rubric. This approach offers significant flexibility: the same model can be adapted to different courses, programming languages, and grading schemes by modifying the prompt alone, without retraining [18]. This characteristic is particularly appealing in academic settings, where resources for model customization are limited.

A crucial distinction in the use of LLMs for grading concerns formative versus summative applications. In formative contexts, LLMs are often used as tutors or

assistants, providing hints, explanations, and debugging suggestions to support learning. Several studies report that students perceive such feedback as helpful and engaging, especially when it explains errors in natural language [17]. In summative contexts, however, the requirements are stricter: grading decisions must be consistent, fair, and justifiable. While LLMs have shown promising alignment with human graders on correctness judgments, they also exhibit variability across runs and sensitivity to prompt formulation, raising concerns about their reliability for high-stakes assessment [15].

From a functional perspective, LLMs enable a form of semantic grading that was largely inaccessible to previous automated systems. Instead of checking only whether outputs match expected results, an LLM can reason about control flow, data structures, and algorithmic intent, potentially recognizing partially correct solutions or alternative valid approaches. This capability directly addresses some of the shortcomings discussed in Sections 2.2 and 2.3, particularly the inability of traditional systems to handle open-ended or design-oriented assignments.

At the same time, the use of LLMs introduces new risks. General-purpose models may generate convincing but incorrect explanations, a phenomenon commonly referred to as hallucinations [14]. They may also implicitly encode biases from their training data, favoring certain coding styles or solution strategies. Unlike test-based graders, whose behavior is fully determined by explicit rules, LLM-based grading systems are probabilistic and opaque, making it difficult to trace grading decisions back to specific criteria [2]. These properties complicate their direct adoption in formal assessment workflows.

In summary, LLMs represent a significant shift in automated code grading. Code-specialized models offer precision and structural awareness but limited explanatory power, while general-purpose LLMs provide flexibility and rich feedback at the cost of reduced transparency and stability. Understanding this trade-off is essential for designing grading frameworks that leverage the strengths of LLMs while mitigating their weaknesses. The following sections examine in detail the reliability, biases, and consistency issues that arise when LLMs are used as evaluators rather than mere assistants.

2.5 Reliability and Consistency of LLM-Based Evaluation

While LLMs enable qualitatively richer forms of automated code assessment, their adoption in academic evaluation critically depends on their reliability and consistency. In educational contexts, especially when assessment outcomes have formal consequences, grading systems must behave in a predictable, stable, and justifiable manner. For LLM-based evaluators, these requirements introduce challenges that differ substantially from those associated with traditional rule-based or test-driven systems.

One central issue is intra-model variability, namely the tendency of the same model to produce different evaluations when presented with identical inputs across multiple runs. This variability arises from the probabilistic nature of LLMs, which generate outputs by sampling from a distribution rather than applying deterministic rules. Even when parameters such as temperature are fixed, non-determinism may persist due to implementation details or stochastic decoding processes [2]. In the context of grading, this behavior undermines reproducibility: a student submission evaluated multiple times by the same system may receive slightly different scores or justifications.

Empirical studies have confirmed the presence of such variability in LLM-based assessment. Jukiewicz reports that while ChatGPT exhibits relatively high repeatability compared to earlier models, non-negligible variation remains, particularly in borderline cases and qualitative explanations [15]. This is especially problematic for partial-credit scenarios, where small changes in reasoning can lead to different numerical outcomes. In contrast, traditional automatic graders, despite their limitations, guarantee strict determinism given the same test environment.

Closely related to intra-model variability is sensitivity to prompt formulation and contextual information. Prompt-based evaluation is a defining feature of LLM-driven grading systems, but it also introduces a source of instability. Minor changes in wording, ordering of instructions, or inclusion of contextual elements such as reference solutions can significantly affect model outputs [18]. From an assessment perspective, this sensitivity raises concerns about robustness: grading outcomes may depend as much on prompt engineering choices as on the quality of the student's code.

Prompt sensitivity also complicates standardization across courses or institutions. Two instructors using the same underlying model but different prompt templates may implicitly enforce different grading criteria, even if they intend to follow the

same rubric. This phenomenon blurs the boundary between grading policy and system configuration, making it difficult to guarantee equivalence of evaluation across settings [24, 1]. As a result, LLM-based grading systems require careful prompt design and validation to avoid unintended inconsistencies.

Beyond individual models, researchers have examined inter-model agreement, that is, the degree to which different LLMs produce similar evaluations for the same submission. Studies comparing multiple LLMs on code-related tasks have shown that agreement is often limited, particularly for qualitative judgments such as code quality or explanation adequacy [10, 24]. While models may converge on clearly correct or incorrect solutions, divergence increases for partially correct or stylistically unconventional submissions. This lack of consensus complicates the notion of an objective “LLM judgment” and suggests that individual model outputs should be interpreted cautiously.

The issue of reliability becomes even more complex when considering model-human agreement. Several recent studies have compared LLM-generated grades with those assigned by instructors or teaching assistants. Results generally indicate moderate to strong correlation on functional correctness, but weaker alignment on qualitative dimensions such as design quality or clarity of explanation [15, 10]. Importantly, discrepancies do not always indicate model error: human graders themselves exhibit significant variability and inconsistency, particularly when grading open-ended programming tasks [19] (see Figure 2.2). This observation challenges the assumption of a single, well-defined ground truth against which LLM performance can be measured.

Nevertheless, differences between human and model judgments have important practical implications. In academic settings, instructors must be able to justify grading decisions to students. When an LLM assigns a score or provides feedback that deviates from human expectations, the lack of transparent reasoning makes it difficult to resolve disputes or explain outcomes. This issue is exacerbated when LLMs generate fluent but loosely grounded explanations, which may appear authoritative despite being only weakly tied to explicit grading criteria.

Taken together, these factors highlight a fundamental tension in LLM-based evaluation. On the one hand, LLMs offer unprecedented flexibility and semantic awareness; on the other hand, their probabilistic and context-sensitive nature challenges traditional notions of grading reliability. For formative assessment, some degree of variability may be acceptable or even beneficial, as long as feedback remains helpful. For summative assessment, however, reliability and consistency are

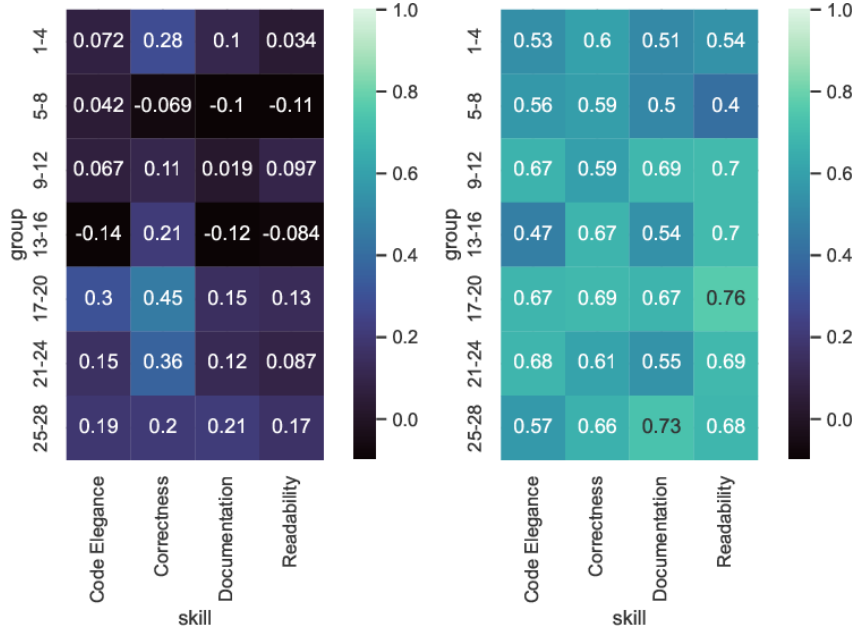


Figure 2.2: Consistency heatmaps, measured with Krippendorff's Alpha (α) (left) and Gwet's AC² (right), for each group of human evaluators and each skill in Messer et al. study. Source: [19].

non-negotiable requirements.

In summary, the reliability of LLM-based grading systems cannot be assumed and must be explicitly measured and controlled. Intra-model variability, prompt sensitivity, inter-model disagreement, and imperfect alignment with human judgment all pose obstacles to their direct adoption in academic evaluation. These issues motivate the need for systematic analysis frameworks capable of quantifying consistency and agreement across models, prompts, and evaluation dimensions, an approach that remains largely underexplored in existing literature and is directly addressed in this thesis.

2.6 Hallucinations and Bias in LLM-Based Assessment

The use of LLMs for automated code assessment introduces a class of risks that are qualitatively different from those of traditional grading systems. Among these, hallucinations and bias are particularly critical, as they directly affect the correctness, fairness, and transparency of evaluation outcomes. In academic contexts, where grading decisions can have significant consequences for students, understanding and mitigating these phenomena is essential.

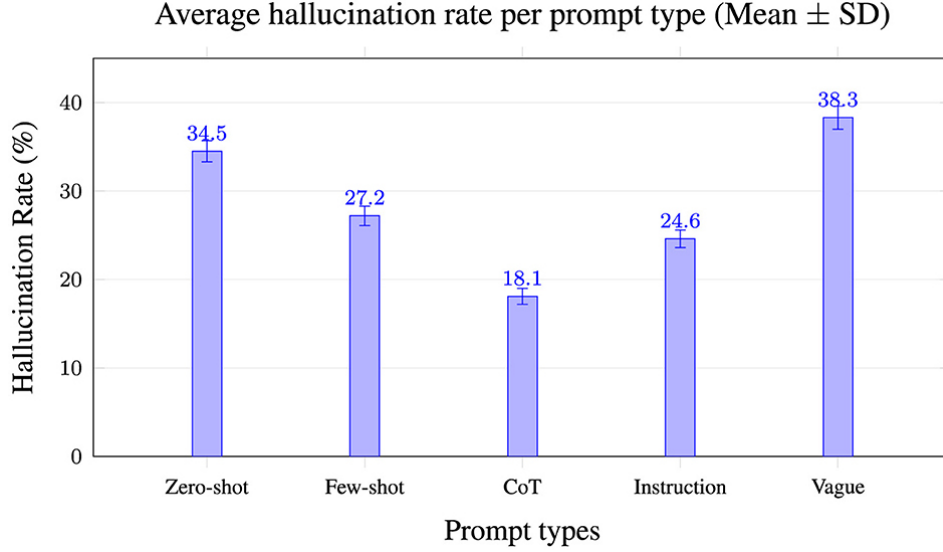


Figure 2.3: Mean $\pm$ SD across 3 seeds $\times$ 5 prompt variants; $n = 100$ examples/model. Chain of Thought (CoT) reduces hallucinations most consistently. Source: [1].

In the context of LLM-based assessment, a hallucination can be defined as a model-generated statement, explanation, or judgment that is not grounded in the actual properties of the evaluated code or in the provided assessment criteria. Unlike simple factual errors, hallucinations are often expressed fluently and confidently, making them difficult to detect without careful inspection [14]. When applied to code grading, hallucinations may manifest as incorrect claims about program behavior, invented bugs, or spurious justifications for a given score.

Hallucinations are particularly prevalent in explanatory outputs, where the model is asked to justify a grade or provide feedback. For example, an LLM may assert that a student’s program fails to handle a specific edge case that is neither required by the assignment nor present in the test inputs. Such explanations can be misleading for students, who may attempt to correct non-existent issues, and problematic for instructors, who must ensure that feedback aligns with the course objectives. Empirical studies have shown that even strong models can produce plausible but incorrect reasoning when prompted to explain their decisions [21, 14, 1]. Figure 2.3 shows how the type of prompt used impacts on the hallucinations phenomenon, highlighting the crucial role of prompt clarity in minimizing hallucinations occurrences.

Hallucinations can also affect grading judgments themselves. In borderline cases, an LLM may assign penalties based on inferred intentions or assumed requirements rather than explicit specifications. This behavior contrasts with traditional graders, whose failures are typically conservative (e.g., missing a valid solution) rather than

inventive. The risk here is not merely inaccuracy, but the erosion of trust: when students or instructors cannot reliably trace grades back to observable code properties, the legitimacy of automated assessment is undermined.

Closely related to hallucinations is the issue of bias. Bias in LLM-based assessment can arise from multiple sources, including training data, prompt design, and evaluation context. Models trained on large corpora of open-source code may implicitly favor certain programming styles, idioms, or libraries, potentially disadvantaging solutions that deviate from these norms despite being correct [2]. In educational settings, this may translate into systematic differences in scoring between stylistically conventional and unconventional student submissions.

Bias may also manifest in score distributions and qualitative judgments. Recent work has shown that LLMs can exhibit consistent tendencies toward leniency or strictness, depending on prompt framing and reference examples [24]. For instance, including a highly optimized reference solution may bias the model toward penalizing simpler but pedagogically acceptable implementations. Similarly, prompts emphasizing “best practices” may implicitly penalize novice-level code in introductory courses, even when such code meets the learning objectives.

Another important dimension of bias concerns explanatory framing. LLM generated feedback may emphasize certain types of issues (e.g., efficiency or elegance) while neglecting others (e.g., correctness or clarity), reflecting biases present in training data or prior usage contexts. This selective attention can skew student perceptions of what aspects of programming are valued, potentially misaligning feedback with course goals [10]. Unlike explicit rubrics, these priorities are not always transparent or controllable.

The combined presence of hallucinations and bias poses significant risks for equity, reproducibility, and transparency. From an equity perspective, biased grading may disproportionately affect certain groups of students or coding styles, raising concerns about fairness. From a reproducibility standpoint, hallucinated explanations and context-dependent biases reduce the ability to obtain consistent results across runs, models, or institutions. From a transparency standpoint, the opacity of LLM decision-making complicates accountability and contestability, both of which are essential in academic assessment.

Recognizing these risks, recent literature has proposed several mitigation strategies. One common approach is grounding, where model outputs are constrained or validated against external sources such as test execution results, static analysis findings, or reference solutions [14]. Another strategy involves structured prompting,

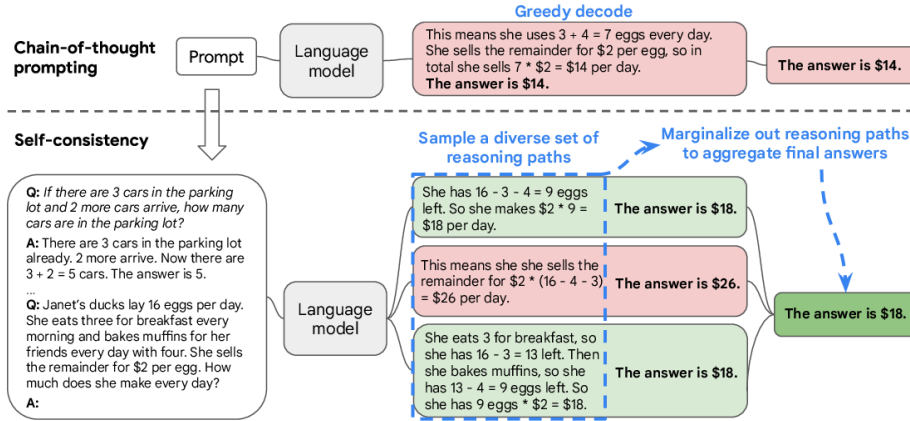


Figure 2.4: Self-consistency method example showing how considering multiple runs reduces hallucinations. Source: [25].

in which the model is required to explicitly reference observable properties of the code or predefined rubric criteria when justifying its evaluation [18]. Ensemble methods, where multiple models or multiple runs are compared to identify unstable judgments, have also been suggested as a way to reduce the impact of individual hallucinations or biases [25] (see Figure 2.4).

Despite these efforts, the literature remains fragmented, with most studies addressing hallucinations or bias in isolation and often in non-educational contexts. There is a lack of systematic, end-to-end evaluations that quantify how these phenomena affect both scores and explanations in realistic academic grading scenarios. This gap motivates the need for frameworks that explicitly analyze and measure hallucinations and bias as first-class properties of LLM-based assessment systems, rather than treating them as anecdotal failures.

2.7 Evaluation of the Generated Explanations Quality

As automated code assessment systems increasingly rely on LLMs, the role of textual explanations has become central to both formative and summative evaluation. Unlike traditional grading systems, which typically provide binary or numerical outcomes, LLM-based approaches can generate natural language justifications that articulate the reasons behind a given score. These explanations are often presented as evidence supporting the evaluation, making their quality a critical factor in determining the usefulness, trustworthiness, and pedagogical value of the system.

In educational contexts, explanations serve multiple purposes. From a formative

perspective, they guide students toward understanding their mistakes, highlighting conceptual gaps and suggesting improvements. From a summative perspective, explanations provide transparency and accountability, allowing instructors and students to inspect how a grade was derived. Prior work in educational feedback consistently emphasizes that meaningful, specific, and actionable explanations are strongly correlated with learning gains, whereas vague or generic feedback offers limited instructional benefit [11, 17].

In the context of LLM-based grading, explanations differ qualitatively from traditional feedback mechanisms. Instead of referencing explicit test failures or predefined rules, LLMs generate explanations by synthesizing patterns learned from data. This enables them to comment on higher-level aspects of code, such as algorithmic strategy, structure, or clarity. However, it also introduces ambiguity: explanations may appear plausible and well-articulated even when they are weakly grounded in the actual code or assessment criteria [14].

A key challenge, therefore, lies in evaluating the quality of generated explanations. Unlike numerical scores, explanations cannot be easily compared using simple error metrics. The literature distinguishes several dimensions of explanation quality, including correctness, relevance, specificity, completeness, and pedagogical usefulness [17]. In code assessment, correctness refers to whether the explanation accurately reflects properties of the student’s program; relevance concerns alignment with assignment objectives; and usefulness captures whether the explanation supports learning or understanding.

Several approaches have been proposed to compare human generated and machine generated explanations. Some studies rely on expert annotation, asking instructors or teaching assistants to rate explanations according to predefined rubrics [10]. Others use proxy metrics derived from natural language processing, such as semantic similarity to reference explanations, overlap with instructor-provided feedback, or coverage of key concepts [26]. While these methods provide partial insights, they are costly, subjective, or difficult to scale.

Automatic metrics borrowed from text generation research, such as BLEU, ROUGE, or embedding-based similarity measures, have also been explored. However, their applicability to educational explanations is limited. These metrics tend to reward surface-level similarity rather than conceptual correctness and may penalize valid explanations that use different phrasing or focus on different but still relevant aspects [22]. As a result, high metric scores do not necessarily correspond to pedagogically effective explanations.

Another line of work focuses on consistency and alignment as indirect indicators of explanation quality. If explanations produced by a model are stable across multiple runs and consistent with assigned scores, they are more likely to reflect an underlying reasoning process rather than ad hoc generation [25]. Conversely, explanations that vary significantly or contradict numerical evaluations raise concerns about hallucination or lack of grounding. This perspective aligns with recent research advocating for joint evaluation of decisions and justifications rather than treating them as independent outputs [13].

Despite these efforts, the literature lacks a unified framework for evaluating explanations in the specific context of automated code grading. Most existing studies focus on isolated aspects, such as correctness or readability, and often evaluate explanations in isolation from the grading process that produced them. This separation is problematic, as the value of an explanation depends critically on its relationship to the corresponding score and assessment criteria. An explanation that is accurate but misaligned with the grading rubric may still be pedagogically misleading.

In summary, while explanations generated by LLMs represent a major advancement over traditional automated feedback, their quality cannot be taken for granted. Evaluating such explanations requires multidimensional criteria that account for correctness, relevance, consistency, and pedagogical intent. The absence of standardized metrics and datasets for this purpose remains a significant gap in the state of the art. Addressing this gap is essential for the responsible adoption of LLM-based assessment systems and motivates the need for structured, end-to-end evaluation frameworks that treat explanations as first-class objects of analysis.

2.8 Critical Synthesis of the State of the Art

The literature reviewed in this chapter highlights a clear and persistent tension between the goals of programming education and the capabilities of existing automated assessment technologies. While significant progress has been made in scaling evaluation and providing rapid feedback, no single approach satisfactorily addresses the combined requirements of correctness, pedagogical value, reliability, and transparency that characterize academic code assessment.

Traditional automatic grading systems, based primarily on test-case execution and output comparison, have proven effective in enforcing functional specifications at scale. However, their limitations are structural rather than incidental. As discussed

in Sections 2.2 and 2.3, these systems reduce assessment to observable behavior under predefined conditions, offering limited insight into students' reasoning, design choices, or partial understanding. Static and dynamic analysis techniques extend the scope of evaluation, but they remain constrained by predefined metrics and struggle to align with high-level educational objectives [5, 17].

The introduction of LLMs represents a qualitative shift in automated assessment. By enabling semantic analysis of code and natural language explanations, LLMs offer capabilities that were previously inaccessible to automated systems. They can generate feedback, reason about algorithmic intent, and evaluate solutions that deviate from rigid templates. Nevertheless, the review of recent work reveals that these advantages come at the cost of new and unresolved challenges. Reliability, consistency, hallucinations, and bias emerge as recurring issues across studies, particularly when LLMs are used in evaluative rather than assistive roles [2, 14, 24].

A critical observation is the lack of systematic evaluation of LLM reliability in academic grading contexts. Most existing studies assess performance through isolated metrics, single runs, or limited comparisons with human graders. While some works report moderate to strong alignment between model-generated and human-assigned scores, they often overlook intra-model variability, inter-model disagreement, and instability across prompts [15, 10]. As a result, current evidence is insufficient to support strong claims about the robustness of LLM-based grading systems, especially in high-stakes summative assessment.

Another major gap concerns the treatment of explanations as first-class assessment artifacts. Although explanations are widely recognized as central to pedagogical effectiveness and transparency, they are rarely evaluated with the same rigor as numerical scores. Existing approaches rely on ad hoc qualitative judgments, proxy similarity metrics, or small-scale human annotation, none of which provide a comprehensive or scalable solution [26, 22]. Consequently, the relationship between scores and explanations, whether explanations genuinely justify the assigned evaluation, remains largely unexplored.

Perhaps most notably, the literature lacks end-to-end frameworks that treat LLMs themselves as objects of systematic evaluation within the assessment pipeline. Existing systems typically focus on using LLMs to grade code, rather than on measuring how well LLMs perform as graders. This distinction is crucial: without explicit mechanisms to analyze consistency, agreement, and failure modes, LLM-based assessment risks reproducing the very opacity and subjectivity it seeks to overcome [19].

Taken together, these observations reveal a clear research gap. While the components required for advanced automated assessment, semantic code understanding, explanation generation, and large-scale evaluation, are individually available, they are not integrated into a unified, methodologically rigorous framework. This gap motivates the work presented in this thesis. Rather than proposing yet another grading tool, the thesis positions itself as a framework for evaluating LLM-based assessment itself, with a particular focus on reliability, consistency, explanation quality, and alignment with human judgment.

In this sense, the contribution of the thesis is not to claim that LLMs are ready to replace human evaluators, but to provide the analytical infrastructure necessary to determine when, how, and to what extent they can be used responsibly in academic contexts. The following chapters build on this synthesis by introducing a framework designed to operationalize these evaluation dimensions in a systematic and reproducible manner.

Chapter 3

Tools and Frameworks

This chapter provides a comprehensive overview of the components that constitute the framework’s infrastructure. It begins by examining the LLMs and the remote inference services that provide the reasoning capabilities of the system. Subsequently, it details the deterministic tools, specifically compilers and testing frameworks, utilized to establish an objective baseline of correctness. Finally, the discussion focuses on the software ecosystem, including the programming environment and the specialized libraries employed for data orchestration, prompt engineering, and statistical validation. By integrating these diverse technologies, the framework achieves a modular design capable of emulating the complex evaluative processes typically performed by human instructors.

3.1 Large Language Models and Inference Services

The primary technological component of this study is represented by LLMs, which serve as the core engines for qualitative code evaluation. These models are sophisticated neural architectures designed to process and generate human-like text by identifying complex patterns within vast datasets. Unlike traditional static analysis tools, LLM possess the unique capability to interpret the semantic intent of source code, enabling the assessment of abstract qualities such as logic flow, structural organization, and adherence to programming best practices.

This section introduces the foundational architectures of these models and provides a detailed technical overview of the specific LLM selected for the research. Furthermore, it defines the communication interfaces, represented by Application Programming Interfaces (APIs) and Software Development Kits (SDKs), that allow these models to be accessed as remote services and integrated into the automated

evaluation environment.

3.1.1 Generative AI Models

The models utilized in this framework belong to the family of Generative Pre-trained Transformers (GPT), which utilize self-attention mechanisms to perform high-level reasoning tasks through token prediction. These models are trained on heterogeneous corpora, including extensive repositories of source code, allowing them to internalize programming paradigms and identify qualitative issues that are typically transparent to compilers.

For the purpose of this study, four specific models from the Gemini and GPT families have been selected to cover a spectrum of reasoning depths and computational efficiencies:

- **Gemini 2.5 Pro:** A high-tier reasoning engine designed for complex, multi-step tasks. It is characterized by an expansive context window, which allows it to process the entire source file alongside extensive pedagogical rules and topic descriptions without losing coherence.
- **Gemini 2.5 Flash-Lite:** An efficiency-optimized version of the Gemini architecture. It is designed for lower latency and reduced computational cost, serving as a benchmark to evaluate the stability of smaller models in qualitative code assessment.
- **GPT-4.1:** A model known for its high-fidelity instruction following. It provides a robust baseline for evaluating the accuracy of the framework’s evidence-extraction logic.
- **GPT-4.1-mini:** A compact version of the GPT-4 architecture, offering a balance between the reasoning depth of the parent model and the speed of a streamlined neural network.

The selection of these models is driven by three primary criteria. First, **cost per token** plays a central role, as one of the intended application scenarios of the framework is academic usage, where large-scale evaluations must remain economically viable. Second, **context length** is a relevant factor, since the evaluation prompts may include multiple artifacts such as student code, reference solutions, and contextual files. Third, **API accessibility** is required to ensure automation, reproducibility, and ease of integration within the framework. Table 3.1 summarizes the key technical and economic characteristics of the selected models as configured in the framework.

Table 3.1: Comparative overview of the LLMs utilized in the framework.

Model Name	Provider	I/O Cost (USD/1M)	Context
Gemini 2.5 Pro	Google	1.25/10.00	1M
Gemini 2.5 Flash-Lite	Google	0.10/0.40	1M
GPT-4.1	OpenAI	2.00/8.00	1M
GPT-4.1-mini	OpenAI	0.40/1.60	1M

These models are treated as automated evaluators capable of identifying specific code segments, defined as “evidences”, that correlate with predefined quality criteria. Their performance is modulated by parameters such as temperature, which is kept consistent across evaluations to ensure that the results of the multi-run stability analysis remain statistically significant.

3.1.2 Application Programming Interfaces (API)

An *Application Programming Interface* (API) is a set of defined rules and protocols that enables different software applications to communicate and exchange data with one another. In the context of this study, APIs serve as the primary gateway to access the reasoning capabilities of the selected models, which are hosted on remote high-performance servers managed by their respective providers.

The API functions as a structured communication channel where the local framework sends a request containing the source code and evaluation instructions, and the remote server returns the model’s generated response. This tool is essential for the framework as it allows for:

- **Remote Inference:** Accessing state-of-the-art neural networks without requiring the local installation of massive computational clusters or specialized GPUs.
- **Standardized Data Exchange:** Ensuring that prompts and outputs are transmitted in a consistent format (typically JSON), which facilitates the subsequent automated parsing of the qualitative feedback.
- **Resource Monitoring:** Tracking usage metrics such as token consumption and execution time, which are necessary for the economic and performance evaluations of the study.

3.1.3 Software Development Kits (SDK)

A *Software Development Kit* (SDK) is a collection of software development tools and libraries that allow for the creation of applications for a specific platform or service. In this research, SDKs represent the implementation layer that translates the abstract protocols of the APIs into functional components within the programming environment. In particular, the framework utilizes the official **openai** and **google-genai** libraries to interface with GPT and Gemini models, respectively. While the API defines the “what” (the endpoints and data structures), the SDK provides the “how” (the actual code used to perform the communication).

The primary function of an SDK in the context of LLM integration is to provide a high-level abstraction over raw HTTP requests. This includes several key capabilities:

- **Authentication Management:** SDKs securely handle API keys and session tokens, ensuring that the connection between the local framework and the remote model providers (such as Google or OpenAI) is authorized and encrypted.
- **Network Resilience:** These kits include built-in mechanisms for managing the inherent instability of cloud-based services, such as automatic retry logic for failed requests and the handling of rate limits imposed by the providers.
- **Data Mapping and Structured Output:** The SDK facilitates the conversion of complex model responses into native data structures. This is essential for enforcing strict adherence to JSON schemas, ensuring that metadata such as token usage and evaluation evidences are captured accurately in order to be able to perform the analyses on the framework’s output.

By utilizing official and community-standard SDKs, the framework ensures a robust and maintainable integration, allowing the research to focus on the evaluative logic rather than the underlying networking infrastructure.

3.2 Compilers and Functional Testing Tools

While the qualitative assessment is delegated to probabilistic models, the framework remains anchored in a rigorous foundation of objective verification. This is achieved through the integration of deterministic analysis tools, which are responsible for validating the structural integrity and functional correctness of the source code. Unlike the interpretive nature of LLMs, these tools provide binary and measurable

feedback, ensuring that the programs are not only well-written from a stylistic perspective but also syntactically sound and functionally compliant.

This section describes the primary components of the framework’s objective evaluation layer: the compiler, used for static diagnostics, and the testing framework, used for dynamic functional validation. Together, these tools establish a formal verification layer. While the functional results can be integrated with the LLM’s qualitative feedback to provide a comprehensive final grade, this functional component remains independent from the core experimental analysis, which focuses strictly on the alignment between generative models and human qualitative judgment.

3.2.1 The GNU Compiler Collection (GCC)

The *GNU Compiler Collection* (GCC) is an integrated suite of compilers for various programming languages, widely recognized as the standard toolchain for C development in Unix-like environments. In this research, GCC serves as the primary tool for static analysis and formal verification of the source code’s structural integrity. Unlike the probabilistic nature of LLMs, a compiler operates through a deterministic pipeline that transforms human-readable code into machine-executable instructions.

The technical role of GCC within the evaluation framework is centered on its diagnostic capabilities:

- **Syntax and Semantic Verification:** GCC identifies violations of the C language standard. While in traditional environments a compilation error terminates the evaluation, in this framework it serves as an objective diagnostic metric. This allows for the assessment of programs that might fail to compile due to minor oversights (e.g., a missing semicolon) but still exhibit high-quality logic and structural soundness, which the LLM can interpret despite the formal error.
- **Heuristic Static Analysis:** By enabling diagnostic flags such as `-Wall` (all warnings) and `-Wextra` (extra warnings), the compiler performs an analysis that goes beyond simple error detection. It flags *code smells* and potentially dangerous constructs, such as uninitialized variables, type mismatches, or unreachable code, which provide an objective measure of a student’s technical precision.
- **Standardization:** GCC allows the framework to enforce a specific version of the C standard (e.g., C99 or C11), ensuring that the code evaluation

is consistent regardless of the environment in which the student originally developed the program.

By quantifying the output of GCC into weighted metrics, the framework establishes a structural baseline that completes the tool’s diagnostic offering, providing additional context to the student alongside the qualitative evaluations of the LLMs.

3.2.2 pvcheck Testing Framework

pvcheck is an automated testing utility specifically designed for the black-box evaluation of small-scale software projects. Its primary function is to verify the functional correctness of a program by comparing its behavior against a predefined set of specifications.

The tool operates by executing the compiled binary and facilitating the following processes:

- **Input/Output (I/O) Comparison:** *pvcheck* automates the feeding of test inputs into the program’s standard input and captures the resulting standard output. It then performs a comparison with the “gold standard” expected results defined by the instructor.
- **Flexible Matching Algorithms:** The tool supports various comparison methods, including exact string matching, numerical comparison with tolerance for floating-point errors, and regular expression matching for dynamic outputs.
- **Functional Scoring:** Based on the ratio of passed to failed test cases, *pvcheck* generates a deterministic report of the program’s functional compliance.

In this study, *pvcheck* provides a deterministic measure of functional compliance. This allows the research to distinguish between a program that works correctly but is poorly written and one that is well-structured but functionally flawed, thereby enabling a multi-dimensional assessment of the student’s work.

3.3 Programming Environment and Software Libraries

The development and execution of the evaluation framework rely on a sophisticated software ecosystem designed to orchestrate the interaction between probabilistic models and deterministic analysis tools. This environment provides the necessary infrastructure for data manipulation, prompt synthesis, and rigorous statistical validation of the results.

This section describes the primary programming language and the specialized libraries that constitute the framework’s technological stack. These components are categorized into three functional layers: the core execution environment, the tools for configuration and text-based orchestration, and the scientific suite used for data analysis and visualization. Together, they ensure that the framework is not only functional but also modular, allowing for the systematic investigation of LLM behavior in a pedagogical context.

3.3.1 The Python Programming Language

Python is a high-level, interpreted, general-purpose programming language characterized by its emphasis on code readability and a vast ecosystem of third-party libraries. In the context of this research, Python serves as the primary orchestrator, responsible for integrating the disparate components of the framework, including LLMs, compilers, and statistical tools.

The choice of Python is dictated by several technical features that are essential for a modular evaluation system:

- **Asynchronous and Multi-provider Support:** Python’s robust support for network communication allows for the efficient management of multiple API calls to different model providers, facilitating the parallel execution of evaluation runs.
- **Extensive Standard Library and Network Utilities:** Modules such as `argparse` for command-line interface management and `re` for regular expression parsing provide the necessary tools for handling user inputs and extracting structured data. Furthermore, the framework integrates the **`requests`** library to perform direct HTTP orchestration with the OpenRouter API. This allows for granular control over request parameters that are not exposed by standard SDKs, such as dynamic provider routing based on real-time pricing and the execution of automated fallback strategies when primary inference services are unavailable.
- **Data-Centric Paradigm:** Python’s dominance in the field of data science ensures seamless interoperability between the framework’s core logic and the advanced mathematical libraries required for the subsequent analysis of model performance.

By utilizing Python as the underlying language, the framework achieves a high

degree of flexibility, enabling the systematic execution of experiments across various models and pedagogical topics.

3.3.2 Templating and Configuration Tools

To ensure that the framework remains generic and adaptable to different academic contexts, the system relies on specialized tools for the dynamic generation of instructions and the externalization of parameters.

- **Jinja2:** A modern and designer-friendly templating engine for Python. It is used to generate the prompts sent to the LLMs by dynamically injecting the student’s source code, the evaluation topics, and the specific structural constraints into predefined text templates. This ensures that the model receives a consistent and well-formatted context for every evaluation.
- **TOML (Tom’s Obvious, Minimal Language):** A configuration file format designed to be easy to read and write. It is utilized to manage the framework’s metadata, including model-specific costs, evaluation weights, and topic definitions. The use of TOML allows for the modification of the framework’s behavior without requiring changes to the core execution logic.
- **Fabric:** An open-source framework for modular prompt engineering. It was utilized as a meta-prompting tool to refine the framework’s system instructions. By applying the `improve_prompt` pattern, the core instructions were iteratively optimized to enforce a deterministic technical persona, ensuring that the LLM’s evidence extraction remains strictly aligned with the required pedagogical criteria and JSON structure.

These tools provide the necessary abstraction to handle the complexity of prompt engineering, allowing for a clean separation between the *what* (the pedagogical content) and the *how* (the execution engine).

3.3.3 Data Analysis and Statistical Libraries

The quantitative evaluation of the framework’s performance relies on a suite of scientific libraries designed for high-performance data manipulation and mathematical optimization. These tools are essential for transforming the raw qualitative outputs of the LLMs into statistically significant metrics that can be compared against human benchmarks.

- **Pandas and NumPy:** These libraries form the foundation for data orchestration within the framework. *Pandas* is utilized for the structured management of datasets, enabling the alignment of model-generated scores and evidences with human-assigned gold standards through complex *DataFrame* operations. *NumPy* provides the underlying support for efficient vectorized operations and numerical computations required during the scoring and calibration phases.
- **Scikit-learn:** This machine learning library is employed to compute key performance indicators, such as the *Mean Absolute Error* (MAE). Within the calibration module, the MAE serves as the objective function to evaluate the accuracy of different weight configurations compared to human judgment. Additionally, it provides utilities for calculating classification metrics, including Precision, Recall, and F1-score, to assess the model's ability to correctly identify specific code conditions.
- **SciPy:** A Python-based ecosystem for mathematics and science. The framework leverages *SciPy* to perform non-parametric statistical tests, specifically calculating the Spearman's rank correlation coefficient. This allows for the validation of the monotonic relationship between AI-generated rankings and instructor evaluations, ensuring that the results are statistically significant.
- **Custom Grid Search Optimization:** To avoid arbitrary weighting, the framework implements a systematic *Grid Search* algorithm. This process explores a discrete parameter space of impact percentages for different criticality levels (high, medium, low) and goodness types (plus, minus). By iteratively testing combinations under pedagogical hierarchy constraints, the algorithm identifies the optimal weight distribution that minimizes the discrepancy with human scoring.

These libraries provide the computational rigor required to move from an anecdotal observation of AI performance to a scientifically validated assessment of its accuracy and reliability in an educational setting.

3.3.4 Visualization and Reporting Frameworks

To facilitate the interpretation of the complex statistical relationships identified during the study, the framework integrates specialized libraries for data visualization.

- **Matplotlib:** A comprehensive library for creating static, animated, and interactive visualizations in Python. It serves as the foundational tool for generating

the graphical representations of the experimental results, providing precise control over the aesthetic and structural elements of the plots.

- **Seaborn:** A Python data visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.

By translating numerical data into visual insights, these tools allow for a more intuitive understanding of the models' strengths and weaknesses, supporting the final conclusions regarding the viability of LLM-driven assessment.

Chapter 4

Implementation

This chapter details the practical implementation of the framework, translating the theoretical components and tools described previously into a functional software architecture. The primary objective is to illustrate how the integration of Large Language Models, deterministic compilers, and automated testing tools is orchestrated to achieve a reliable and scalable assessment system.

The discussion begins by establishing the **Design Goals**, defining the engineering principles and the pedagogical objectives that guide the development of the framework. Subsequently, the **Logical Architecture** is presented, decomposing the system into functional layers to ensure a clear separation of concerns between data ingestion, evaluation, and result reporting. The following sections provide a granular analysis of the execution flow, the prompt engineering strategies, and the abstraction layers used to manage multi-provider model interactions, concluding with the processing pipeline used for experimental validation.

4.1 Design Goals of the Framework

The primary goal of the framework is to evaluate *how* and *to what extent* LLMs, when used through a controlled configuration based on prompt engineering and structured output constraints, are effective in an academic code assessment setting. Effectiveness is investigated from both a *formative* perspective, in terms of the quality and usefulness of the feedback provided to students, and a *summative* perspective, in terms of the reliability and coherence of the numerical evaluations produced by the models.

Given the intrinsic difficulty of fully generalizing automatic assessment frameworks across heterogeneous courses and programming tasks, *checkmyc* is designed to

be *course-specific but extensible*. The framework does not aim to define a universal grading policy; instead, it enables the customization of evaluation topics, assessment criteria with their weightings, and score ranges through editable prompt templates. The experimental evaluation presented in this work focuses on the introductory programming course “Fondamenti di Informatica” taught at the first year of the Degree in Computer and Electronic Engineering at the Department of Industrial, Computer and Biomedical Engineering of the University of Pavia, which targets the C programming language. This choice is motivated by the availability of an existing curated dataset of student submissions and human evaluations. The same framework can, in principle, be adapted to other programming courses that require the assessment of a single program submission by modifying the evaluation topics prompts.

Beyond the primary experimental objective, the framework is guided by three secondary design goals:

1. **Course-level personalization** allows instructors to explicitly encode course-specific expectations, evaluation topics, and grading weights, avoiding the implicit assumptions often embedded in generic LLM-based tools.
2. **Student support** is addressed by providing structured, topic-oriented feedback that can be used for self-assessment and exam preparation without the delays inherent to direct instructor interaction.
3. **Instructor support** is considered by positioning the framework as a complementary tool to traditional automatic graders, capable of capturing qualitative aspects of code correctness and design that are typically ignored by output-based testing alone.

The intended users of *checkmyc* are therefore both students and instructors, each benefiting in a distinct but complementary way. For students, the framework serves as an automated self-evaluation tool that can accelerate the learning process by providing immediate, structured feedback aligned with the course requirements. For instructors, it represents a potential auxiliary instrument that, depending on its measured effectiveness, can support or partially compensate for the limitations of purely functional automatic grading systems, especially in scenarios where manual review of all submissions is impractical.

Importantly, the framework is explicitly designed for *experimental evaluation* rather than for the direct production of official exam grades. No model fine-tuning or prompt-tuning strategies are employed. Instead, the behavior of LLMs is studied

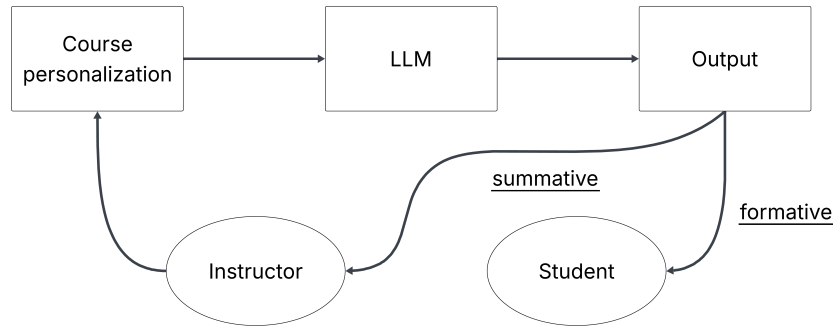


Figure 4.1: High-level overview of the design goals of the framework and its intended users.

under controlled prompt-engineering conditions, with the goal of analyzing their performance when used “as-is”. The evaluation focuses on three complementary dimensions: agreement with human judgments in terms of correctness, coherence across different models when assessing the same submissions, and internal consistency of repeated evaluations performed by the same model. This design choice allows the framework to serve as an experimental testbed for analyzing strengths and limitations of LLMs in academic assessment contexts, rather than as a prescriptive grading authority. Figure 4.1 provides a high-level overview of the design goals and target users of the framework, highlighting the relationship between experimental objectives, personalization mechanisms, and stakeholder roles.

4.2 Logical Architecture

The framework is composed by the *checkmyc* program and the *Analysis Component*. The program *checkmyc* and is structured around a modular logical architecture designed to clearly separate data acquisition, prompt construction, evaluation, aggregation, and result presentation. This separation is intended to improve clarity, extensibility, and experimental control, while avoiding tight coupling between heterogeneous evaluation mechanisms. At a high level, the architecture can be decomposed into five main layers: an *Input Layer*, a *Prompt Layer*, an *Evaluation Layer*, an *Aggregation Layer*, and an *Output Layer*. Figure 4.2 illustrates the logical architecture of the framework, highlighting the main layers, their interactions, and the parallel evaluation paths.

The **Input Layer** is responsible for collecting and normalizing all artifacts required for the evaluation process. These include the student program to be assessed,

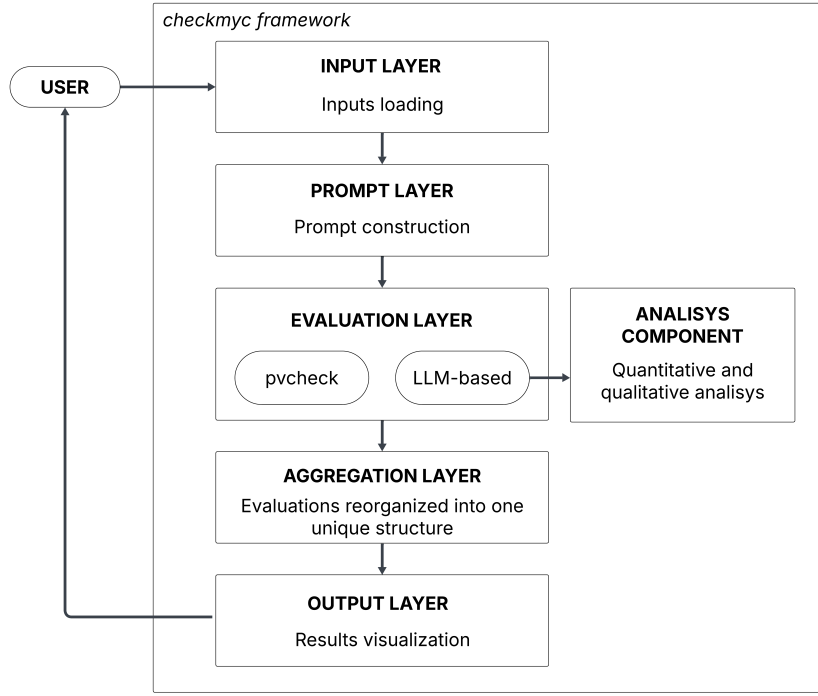


Figure 4.2: Logical architecture of the framework.

a set of evaluation topics, a reference program, and the program context (the exam text). While the core of the evaluation is driven by pedagogical topics and their specific conditions, the reference program and context serve as essential support elements. They provide the LLM with a course-specific standard and environmental constraints, ensuring that the evaluated conditions are analyzed within the correct functional and stylistic frame of reference.

These artifacts are processed by the **Prompt Layer**, where they are dynamically reorganized to build the final instructions for the LLM. This layer utilizes the collected data to synthesize a prompt that guides the model through a structured qualitative assessment.

The **Evaluation Layer** performs the actual assessment through two parallel and conceptually independent paths:

- **Qualitative LLM Evaluator:** This component analyzes the student code against the provided topics and conditions. It identifies textual “evidences” and maps them directly to numerical penalties based on a predefined weighting scheme, producing structured feedback (see Section 4.8).
- **Deterministic Functional Grader:** Based on *pvcheck* and secondary tests (compilation and time execution), this component checks the program to assign

a functional correctness score. While this functional grade is included to provide a comprehensive overview of the student’s work, it remains secondary and independent from the qualitative analysis performed by the LLM and the related final analysis.

The outputs are then managed by the **Aggregation Layer**, which organizes scores, textual explanations, and metadata into a unified representation. A central element of this process is the structured JSON schema, which defines a stable interface between evaluation and analysis. This layer allows for the calculation of a final grade that can optionally combine LLM and functional results through configurable weights, providing a holistic view of the submission.

The **Output Layer** presents these results in two formats: a machine-readable JSON file for data persistence and a human-readable HTML report for student feedback and instructor review. Finally, the framework includes an external **Analysis Component**. This component operates exclusively on the structured outputs to conduct the quantitative investigations, such as stability and human-alignment tests, presented later in this work.

4.3 System Model and Formal Notation

To ensure a rigorous and consistent description of the framework’s behavior, this section establishes the formal notation used throughout the chapter. The model defines the relationship between pedagogical objectives, evaluation conditions, and the numerical parameters that govern the scoring logic.

4.3.1 Evaluation Entities

The assessment universe of *checkmyc* is structured as a hierarchy of two primary entities:

- **Topics (τ):** The set of high-level assessment subjects (e.g., Modularity, Error handling). This set of topics is denoted as $\tau = \{T_1, T_2, \dots, T_n\}$, where n is the total number of topics defined for a specific course. Each topic T_i obtains a score value S_{T_i} from the evaluation, whose maximum value is defined as max_i , and has an associated weight W_{T_i} , which impacts on the single LLM-related score.
- **Conditions (C):** Each topic T_i contains a set of m discrete conditions $C_{i,j}$ (where $j = 1, \dots, m$) that the LLM must detect within the code. Each condition

Table 4.1: Notation and valid domains for condition subscripts.

Variable	Values	Description
g	$\{+, -, =\}$	Condition nature (Positive, Negative or Neutral).
c	$\{l, m, h, n, d\}$	Severity level (<i>low, medium, high, neutral, decisive</i>).
i	$1, \dots, n$	Index of the evaluation topic.
j	$1, \dots, m$	Index of the specific condition within a topic.

is defined together with a pair of associated qualitative attributes: *goodness* and *criticality*. We denote $C_{i,j}^{g,c}$ as the j -th condition of the i -th topic, where g and c represent its goodness and criticality. Depending on this pair, each condition $C_{i,j}^{g,c}$ has a specific weight $w_i^{g,c}$ for the score computation of topic T_i ; the relative scoring logic is detailed in Section 4.8

The valid domain for the defined indices is defined in Table 4.1, while the pair validity of the two attributes is shown in Table 4.2.

These conditions are derived directly from the instructor’s expertise and are designed to formalize a personal grading methodology. Rather than requiring the model to evaluate code according to generic standards, these conditions serve as the primary vehicle through which the LLM “enters” the specific evaluative logic of the professor. By translating a subjective pedagogical approach into a set of discrete, verifiable rules, the framework ensures that the automated assessment reflects the formal (rather than purely literal) judgment of the human evaluator.

Table 4.2: Validity of Goodness-Criticality pairs.

Pair (Goodness, Criticality)	Validity	Scoring Role
$(+, \{low, medium, high\})^*$	Valid	Contributes to score increment.
$(-, \{low, medium, high\})^*$	Valid	Contributes to score decrement.
$(-, decisive)$	Valid	Forces topic score to 0.
$(=, neutral)$	Valid	Informational; no score impact.
$(+, decisive)$	Invalid	Positives cannot be <i>decisive</i> .
$(\{+, -\}, neutral)^*$	Invalid	Neutral reserved for goodness =.

* Every condition defined by only one element within the braces.

4.3.2 Weighting Schemes: Percentage vs. Fixed

The framework supports two distinct definition modalities for $w_i^{g,c}$, allowing the instructor to choose the most appropriate scaling for each specific course:

1. **Percentage-based Weighting:** The impact is expressed as a percentage $p_i^{g,c}$ of max_i . The resulting formula is:

$$w_i^{g,c} = \frac{max_i \cdot p_i^{g,c}}{|C_i^{g,c}|} \quad (4.1)$$

where $|C_i^{g,c}|$ is the total number of conditions of type (g, c) defined in the i -th topic’s prompt.

This way, for instance, the condition type $(-, high)$ can be configured to subtract 60% of max_i .

2. **Fixed Weighting:** A single numerical value is assigned to a specific condition type:

$$w_i^{g,c} = w_{i, fixed}^{g,c} \quad (4.2)$$

With this modality each detection of the condition (g, c) for the topic T_i adds or subtracts a fixed amount of points, regardless of the topic’s overall scale.

Informational conditions $(=, neutral)$ are used to notify the student about non-impactful code aspects, providing qualitative feedback without affecting the grade. Conversely, decisive conditions $(-, decisive)$ act as a “circuit breaker”: if detected, they immediately override any positive findings and set the topic score to 0. For these implementation choices these two conditions types are not associated to a configurable weight. The mathematical integration into the final topic score are formally defined in Section 4.8.

4.4 Execution Pipeline and Workflow

This section describes the end-to-end execution flow of the *checkmyc* command-line program, from initial input parsing to the generation of the final structured reports. The implementation¹ is organized as a linear pipeline where the overall control flow is deterministic, while the LLM inference step introduces the only non-deterministic component.

¹*checkmyc*: <https://github.com/FedeRoma01/checkMyC.git>

Phase 1: CLI Parsing and Runtime Parameters. Execution begins in the main function of `__main__.py` by parsing command-line arguments via `argparse`. The inputs are categorized into:

- **Evaluation Inputs:**
 - student C program(s);
 - reference solution;
 - context file (exam text);
 - input file to enable the deterministic tests’ execution;
 - parameter to enable percentages weightings.
- **LLM Settings:**
 - model to be used for the qualitative evaluation;
 - model provider;
 - temperature hyperparameter;
 - input and output costs limits.
- **Path and Output Configuration:**
 - parameter defining whether to use absolute or relative predefined paths;
 - destination directory for artifacts.

All these parameters are specified in the `README.md` file, together with their explanations.

Phase 2: Configuration Loading and Path Resolution. The framework initializes the global state by loading the `config.toml` file. This phase resolves all relevant paths for prompts and datasets through dedicated helpers in `config.py`. It also loads the LLM metadata (pricing and model options) and the weights used for scoring from the `llm.toml` configuration. A detailed description of the customizable configuration is provided in Section 4.5

Phase 3: Program Discovery and Context Assembly. The system supports both individual file assessment (for students) and batch processing of directories (for instructors). The `programs_loading` function normalizes these inputs into a list of targets. Context is assembled from either an explicit `-exam` directory, containing problem statements and *pvcheck* configurations, or from individual files passed via CLI flags.

Phase 4: Deterministic Evaluation Layer. Before any probabilistic inference occurs, *checkmyc* executes a suite of deterministic checks. As implemented in `evals.py`, this includes:

- **Compilation and Static Analysis:** Utilizing GCC with strict flags to detect syntax errors and warnings.
- **Functional Testing:** If the necessary resources are present, the framework invokes *pvcheck* to execute the student program against a test suite.

These checks produce an objective metrics vector that allows to distinguish the qualitative aspect from the functional one of the evaluated program and to visualize both in the output.

Phase 5: Prompt Rendering and Structured Output Enforcement. The framework prepares the LLM instructions by loading the student program and annotating it with line numbers to ensure precise evidence referencing. A rendering context is built containing the JSON schema, topic descriptions, and the reference solution. System and user prompts are then compiled from Jinja2 templates, ensuring that the model is constrained by the predefined output schema generated in `config.py`. This process is further elaborated in Section 4.6, which focuses on the prompt engineering and enforcement strategies.

Phase 6: LLM Invocation and Model Dispatching. The actual model call is routed through a provider-aware dispatcher in `model_runner.py`. Depending on the configuration, the request is sent to the corresponding API wrapper (OpenAI, Google, or OpenRouter). All providers are instructed to return a structured JSON response that complies with the schema defined in the Prompt Layer. The specifics of this phase are detailed in Section 4.7.

Phase 7: Cost Estimation and Score Aggregation. Upon receiving the response, token usage is normalized and a call-cost estimate is computed based on the pricing metadata. This computation needs the user to specify the three token-types costs inside the `llm.toml` configuration file for the model intended to be used. The framework then aggregates the results into three tiers: an objective functional score, a weighted qualitative LLM score based on the identified “evidences”, and an optional combined final score. This phase, together with the following one are treated in Section 4.8.

Phase 8: Output Materialization. The final stage of the pipeline involves the generation of a comprehensive JSON artifact, designed to encapsulate all relevant metadata, the complete LLM output, and the resulting aggregated scores into a single structured file. A HTML report is simultaneously generated from this JSON via a template, providing a human-readable interface for students and instructors without altering the underlying data.

4.5 Configuration Mechanisms

The architectural flexibility of the framework is primarily achieved through a trio of `toml` configuration files. This design choice ensures a strict separation between the core execution logic and the pedagogical parameters, allowing the system to be adapted to different academic courses and grading policies without modifying the underlying Python code. The mathematical relationship between these parameters and the final grades obtained in the output is detailed in Section 4.8.

4.5.1 System Orchestration

The `config.toml` file serves as the central hub for environmental settings and global weighting. It defines the directory structure for source codes, outputs and topics. A critical feature of this file is the definition of the balance between deterministic metrics (unit tests) and qualitative LLM evaluations, as shown in Listing 4.1, used to obtain an optional final unique score that takes into account both functional and qualitative evaluation aspects.

Listing 4.1: Global weighting in `config.toml`

```
# Paths configuration and final score balance

[paths]
# specific configuration toml files
questions = "src/checkmyc/config/questions.toml"
llm = "src/checkmyc/config/llm.toml"
# pre-configured paths
output_path = "final_output"
programs_path = "resources/sources"
exam_text_path = "resources"
sys_prompt_path = "src/checkmyc/data/prompts/system"
usr_prompt_path = "src/checkmyc/data/prompts/user"
topic_prompts_path = "src/checkmyc/data/prompts/topics"

[combined_weights]
llm = 4.0      # Weight for qualitative analysis
tests = 6.0    # Weight for functional correctness (pvcheck, gcc)
```

4.5.2 Pedagogical Rules and LLM Scoring

The `llm.toml` file represents the core of the framework’s “evaluative intelligence”. It allows for a granular definition of the assessment rubric.

For each topic T_i , this file presents the following customizable elements (see Listing 4.2):

- W_i (weight): the topic’s weight for the single LLM-related score that takes into account all the n defined topics.
- **prompt filename** (filename): the name of the file containing the topic’s prompt.
- max_i (max_score): the maximum score that the topic can reach.
- $w_i^{g,c}$ (high, medium, low): both the fixed (first element in the list) and percentage (second element in the list) weights for all the valid and score-impactful $C_{i,j}^{g,c}$.

Listing 4.2: Topic and weight definition in `llm.toml`

```
[[topics]]
name = "Modularity"
weight = 1.0
filename = "mod.md"
max_score = 10

[topics.weights.minus]
high   = [-1.0, 0]      # [weight coefficient, impact percentage]
medium = [-1.0, 100]
low    = [-1.0, 0]

[topics.weights.plus]
high   = [0, 40]
medium = [0, 30]
low    = [0, 30]
```

4.5.3 Objective Testing and Requirements

Focused on the deterministic side of the assessment, `questions.toml` (see Listing 4.3) manages the weights for specific exam questions (used for the `pvcheck` related score) and objective metrics like compiler warnings and performance.

4.5.4 Course-Level Adaptability

The combination of these files enables a “plug-and-play” approach to course customization. By adjusting these three configuration files, the framework can reflect

the most appropriate assessment method for the specific course, balancing functional and qualitative evaluation aspects. This modularity ensures that the LLM operates within the specific boundaries and grading philosophies established by the human instructor.

Listing 4.3: Objective test weights in questions.toml

```
[questions_weights]
1 = 1.0
2 = 1.0
...

[tests_weights]
warning = 2.0
performance = 2.0
pvcheck = 6.0
```

4.6 Prompt Engineering and Structured Output

A central component of the *checkmyc* framework is the construction of prompts and the enforcement of a strictly structured output format. This section describes the system prompt, user prompt, and topic-specific prompts, how they are combined and rendered at runtime using the Jinja2 engine, and how a predefined JSON schema is used to constrain the model output. Together, these mechanisms act as a controlled experimental interface, ensuring comparability, reproducibility, and the ability to perform automated analysis on the evaluations.

4.6.1 System and User Prompts

The prompting strategy follows a clear separation between a **system prompt** and a **user prompt**, each fulfilling a distinct role in the evaluation pipeline.

The **system prompt** serves as the foundational configuration, establishing the global role, the evaluative philosophy, and a rigorous set of constraints. Unlike the user prompt, it is not based on user input or customisation, so it is not thought to be modified with the exception of the *Role* in the *Persona Alignment*. However it is part of the final prompt that instructs the LLMs behaviour. Its architecture is organized into the following functional modules, refined through the *Fabric*'s pattern `improve_patterns`, for high-fidelity instruction:

- **Strict Requirements and Structural Guardrailing:** This section implements a zero-tolerance policy regarding output format by enforcing a JSON-only

constraint and forbidding any external commentary or thought-process exposure. By treating the LLM as a stateless functional component, it ensures that the output is programmatically consumable by the framework’s aggregation logic without parsing errors.

- **Persona Alignment (Role and Objective):** The model is cast as an *expert C programmer acting as a deterministic, rule-based evaluator* for beginner student code. This role calibrates the model’s technical depth, ensuring it focuses on fundamental algorithmic correctness and adheres to a philosophy of non-subjective interpretation.
- **Input Data Categorization:** This module explicitly defines the four pillars of the inference context: the Evaluation Topics, the Context Section (requirements), the Reference Program (gold standard implementation), and the Evaluated Program (student code). This structure ensures the model understands the hierarchy of information provided in the user prompt.
- **Evaluation Process and Evidence Generation:** This is the most critical functional module, defining the *Rules of Engagement* for triggering conditions specified in the input topics. This module is entirely shown in Listing 4.4. Inside it the structure of a single *evidence* is defined; it represents the “molecular” part of the output generation.

Listing 4.4: Evidence Generation Rules

```
## EVALUATION PROCESS

### Evidence Generation

For each topic, generate an array of evidences strictly
  based on code patterns that directly trigger the provided
  Evaluated Conditions.

- Trigger Definition: A condition is triggered ONLY if the
  behavior described is PRESENT.
- Global Code Scanning: Every evaluated condition must be checked
  against the entirety of the 'Evaluated Program'. Evaluation is
  only complete when all possible triggers in all functions have
  been identified and consolidated.
- No Early Exit: Finding one manifestation of a condition does not
  terminate the search for that specific 'condition_id'.
- Explicit Manifestation: Only produce evidence if the student's
  code explicitly triggers a condition as defined.
```

- Positive Trigger Only: Produce evidence ****ONLY**** when the code behavior matches the condition's description. If the code does the opposite of what is described, it is a non-trigger and must be ignored.
- Silence rule: If a code behavior is the **OPPOSITE** of a condition, or simply does not match it, do not mention it. An evidence must be a **PROOF** of the condition's existence, never a description of its absence or its successful avoidance.
- Binary Satisfaction: Conditions must not be triggered for "partial" successes. If a condition describes a specific strategy, all components of that strategy must be present in the code to trigger the condition.
- The "All" Clause (Universal Quantification): An evaluated condition that starts with "All" must be triggered ****only if**** every applicable instance within the identified scope strictly satisfies the requirement without exception.
- Contextual Interpretation: Use the 'Context', if present in the topic, to interpret domain-specific terms exactly as specified.
- Line Context: In the 'lines' array, include all strings (individual lines or ranges) that provide strictly-necessary context for the identified trigger.
- No Inference of Absence: Do not generate evidence regarding the absence of condition.
- Scope Identification: Each evidence's comment must explicitly identify its scope (e.g., function name, code block) where the condition applies, using backticks (e.g., "In 'function_name', ...").

Evidence Object Fields

- "condition_id": The exact integer ID associated with the triggered condition as listed UNDER the current topic name.
- "comment": A concise, impersonal technical description of the code behavior triggering the condition.
- "lines": JSON array of strings representing the exact line numbers or continuous ranges (e.g., ["5"], ["12-15"]).

- **Output Schema Specification:** The final section provides an explicit template for the JSON response (see Listing 4.5). It mandates the use of exact key names (topic_name, condition_id, comment, lines) to ensure a stable interface between the probabilistic model generation and the deterministic scoring logic implemented in `evals.py`. This section is included in the prompt even though the relative API functionality is being used, as some models do

not support it. In such cases, this section acts as a substitute to instruct the model to use the output schema.

Listing 4.5: JSON schema

```
{
  "evaluations": [
    {
      "topic_name": "Topic Name",
      "evidences": [
        {
          "condition_id": Integer,
          "comment": "String",
          "lines": ["String"]
        }
      ]
    }
  ]
}
```

The **user prompt**, managed via the `generate_user_prompt` function in `config.py`, provides the concrete material for analysis. While the system prompt defines the *how*, the user prompt supplies the *what*, populating the input elements described in *Input Data Categorization* with course-specific content.

To manage this dynamic injection, the framework employs the **Jinja2 templating engine**. The user prompt is defined as an external Markdown template file containing specific **placeholders** (e.g., `{{ topics }}`, `{{ program }}`) that are resolved at runtime. This choice decouples the prompt's structural design from the Python logic, allowing the system to:

- **Render complex structures:** the engine programmatically iterates over evaluation topics, conditions, and code snippets, ensuring consistent formatting regardless of the input size or the number of criteria.
- **Maintain architectural separation:** the system prompt remains a static control variable, while the user prompt is generated at runtime by rendering external templates through the injection of raw data.

At execution, Jinja2 resolves all placeholders and logic, producing a flattened, context-rich string. Specifically, the engine ensures that disparate data sources,

such as student code and gold-standard implementations, are isolated using clear **Markdown delimiters**, such as fenced code blocks and headers. This ensures that the model receives a clean, structured input where the hierarchy between requirements, reference code, and student implementation is explicitly preserved for the inference process.

4.6.2 Topic-Specific Prompts

The user prompt, as previously said, contains the evaluated topics. Each topic T_i (defined in a markdown file) describes a single assessment dimension and specifies the conditions the model must verify. These topics prompts share a common internal structure:

- **Objective:** An explicit description of the specific aspect of the program being evaluated.
- **Topic Context:** Detailed instructions that guide the model on how to interpret domain-specific terms and code patterns. This section defines what should or should not be considered for the specific topic (e.g., ignoring functional correctness to focus on structure) and provides definitions for internal logic and in relation to defined conditions in order to avoid their misinterpretations.
- **Evaluated Conditions:** The list of code-related conditions to be detected, following the rules specified in the *Evaluation Process and Evidence Generation* within the system prompt.

Each condition $C_{i,j}^{g,c}$ is strictly identified by a unique **condition id** within T_i . An example of conditions definition is shown in Listing 4.6.

All selected topics are processed within a single model invocation. The list of topics determines both the content of the prompt and the structure of the output JSON, ensuring that the model reasons over a unified and structured set of criteria. For the considered study case, five topics have been selected ($n = 5$):

- **Modularity:** Evaluates code decomposition, function reuse, and the logical organization of the solution.
- **Correct use of dynamic memory:** Focuses on safe heap management, including allocation checks and the systematic prevention of memory leaks.
- **Appropriate data structure:** Assesses the selection and implementation of data structures (e.g., structs, arrays) in relation to the problem requirements.

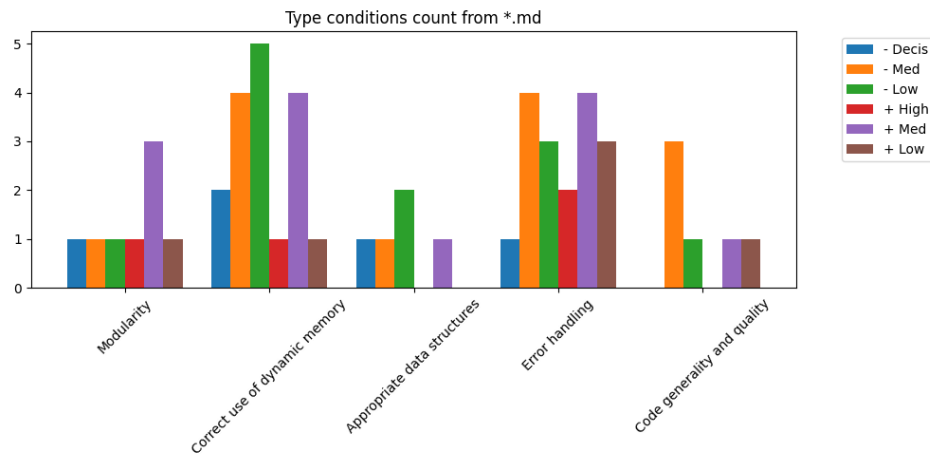


Figure 4.3: Frequency of defined condition types across the used topics

- **Error handling:** Examines the program’s robustness in managing runtime failures, such as I/O issues or invalid user input.
- **Code generality and quality:** Checks the overall code quality in terms of input handling, formatting consistency, and support for understandability.

Listing 4.6: Example of evaluated conditions definition

1. The reading file function can return NULL pointer to the caller due to not checking and handling allocation failures (goodness: -, criticality: low).
2. The number of command-line arguments is not checked (goodness: -, criticality: medium).

Condition Distribution across Topics

The configuration of the evaluation framework is strictly linked to the nature and number of conditions defined within each topic’s prompt. An analysis of the pedagogical requirements for the target course led to the definition of a specific set of conditions, whose distribution across the five assessment topics is illustrated in Figure 4.3.

The chart highlights a structural non-homogeneity in the prompt design, which reflects the instructor’s grading priorities:

- **Prevalence of Medium-Impact Rules:** Most topics rely on a high number of *medium-criticality* conditions (both positive and negative) to capture granular implementation details.

- **Criticality Gaps:** Certain topics, due to their technical nature, lack specific criticality tiers. Notably, the absence of any defined *high-criticality negative* conditions $(-, high)$ across all topics is a deliberate choice in the current prompt version.
- **Topic Complexity:** Topics like Error Handling or Correct use of dynamic memory show a higher density of negative conditions compared to others, aiming to strictly penalize critical failures in those domains.

This a-priori distribution of conditions represents the “logical blueprint” of the evaluator. It directly constrains the subsequent weight calibration phase, as the optimization of percentage weights is only possible for the condition types effectively present in the prompts. Furthermore, the sparsity of certain tiers justifies the adoption of derived weights for the less frequent categories to avoid overfitting during the grid search.

4.6.3 Structured Output Schema

To enable systematic analysis and ensure that qualitative feedback can be programmatically processed, *checkmyc* enforces a strictly structured output format. This is achieved by leveraging the **Structured Output** feature provided by modern LLM APIs (such as those from OpenAI and Google), which forces the model to strictly adhere to a user-defined JSON schema during the generation process.

A critical aspect of this enforcement is that the schema is **dynamically generated** at runtime by the `generate_schema` function in `config.py`. By injecting the exact names of the requested pedagogical topics directly into the schema’s validation logic, the framework creates a deterministic constraint that the model cannot bypass and allows to automatize the schema creation, without the needing of manually change it when different topics are used. This dynamic binding ensures that:

- **Topic Consistency:** The model is restricted to using only the predefined topic names as keys, preventing “label drift” where the LLM might otherwise invent or approximate categories.
- **Post-processing Integrity:** Every field in the resulting JSON object is guaranteed to align with the framework’s internal evaluation logic. Without this level of stringency, any deviation in element naming would cause the parsing logic to fail, effectively preventing the production of the final report and precluding subsequent statistical analysis.

- **Structural Hierarchy:** The schema (whose structure is shown in Listing 4.5) mandates an array of evaluations, where each entry must include a `topic_name` and a list of evidences containing a `condition_id` (integer), a `technical comment` (string), and a `lines` array (strings).

By treating the schema as an immutable contract for each specific execution, *checkmyc* ensures that the probabilistic output of the LLMs is transformed into a reliable, machine-readable data point suitable for both student feedback and rigorous academic investigation.

4.7 Multi-Provider Abstraction Layer

To ensure the framework remains model-agnostic and resilient to changes in specific API implementations, *checkmyc* adopts a provider-based abstraction pattern. This layer decouples the core evaluation logic from the low-level details of each SDK, allowing for seamless switching between models like GPT-4, Gemini, and those hosted on OpenRouter.

4.7.1 The Provider Pattern Strategy

The architectural cornerstone of this layer is the `BaseProvider` abstract class in `api/utils_api.py`. This class defines a mandatory interface that every concrete provider must implement, centered around two primary methods:

- `run()`: Handles the actual API request, passing the system and user prompts along with the JSON schema.
- `normalize_usage()`: Standardizes the token usage metadata returned by different vendors into a unified format for cost calculation.

Concrete implementations, such as `OpenAIProvider` and `GeminiProvider`, wrap their respective SDKs. The `get_provider` factory function dynamically instantiates the correct class based on the `-provider` flag passed at runtime, ensuring the orchestrator remains unaware of the specific library being used.

In the case of OpenRouter, the framework implements a dual-mode strategy depending on the specificity of the provider requested:

- **OpenRouterProvider:** Utilized when a specific underlying provider is identified through the OpenRouter gateway. This implementation leverages the **OpenAI SDK** by overriding the `base_url` to OpenRouter's endpoint, providing a familiar interface for standardized calls.

- **OpenRouterRequestProvider:** Utilized when a generic provider is specified or when the user intends to leverage advanced routing parameters, such as **cost limits** (`prompt_max_price` and `completion_max_price`). This mode bypasses the SDK in favor of a direct **HTTP request** (`requests.post`), allowing for the injection of a custom JSON payload that includes provider sorting by price and fallback permissions.

4.7.2 Robustness and Error Handling

Given the inherent instability of network-based APIs, the abstraction layer implements an error-classification system:

- **TransientAPIError:** Covers temporary issues like rate limits (429) or server timeouts (5xx). The framework handles these through an exponential backoff retry logic, attempting the call up to three times before declaring a failure.
- **FatalAPIError:** Triggered by unrecoverable issues such as invalid API keys or malformed requests (400). These errors cause the framework to halt the current evaluation and log a critical failure.
- **JSON Handling:** If the model used does not support structured output, it could return an invalid JSON string despite the structured output enforcement present in the system prompt, and the `save_debug_pair` function captures both the raw output and the context metadata for later inspection.

4.7.3 Experimental Control and Determinism

The abstraction layer is responsible for enforcing experimental constraints across different models, ensuring that comparisons are conducted under identical stochastic settings. Parameters such as the temperature are passed through the provider interface to maintain consistency in the degree of randomness during generation.

To optimize the evaluation process, particularly during batch processing of multiple student submissions, the framework implements advanced prompt caching strategies tailored to the specific capabilities of each provider. These mechanisms significantly reduce both latency and operational costs by avoiding the redundant processing of static context.

- **Implicit OpenAI Caching:** In the `OpenAIProvider`, the framework leverages an automated caching system for prompts exceeding 1024 tokens. Requests are routed based on a hash of the prompt’s initial prefix. As implemented in

`openai_api.py`, the framework explicitly provides a `prompt_cache_key` and utilizes the `in_memory` retention policy. This ensures that common prefixes, such as the System Prompt and Context Section, remain active in GPU memory for up to one hour, improving hit rates for subsequent evaluations.

- **Explicit Gemini Caching:** In the `GeminiProvider`, a more granular, manual caching strategy is employed (see Listing 4.7). Upon the first API call, if multiple programs are being evaluated (`cache_flag`), the framework isolates the static portion of the user prompt, containing the requirements and reference solution, from the student’s code using a specific separator. This static context is then used to create an explicit cache via `client.caches.create` with a defined Time-To-Live (TTL). Subsequent requests in the same session refer to this cached content, transmitting only the student’s unique code.

Listing 4.7: Gemini explicit caching creation

```
# Create a cache with a 5 minute TTL (300 seconds)
self.cache = self.client.caches.create(
    model=model,
    config=types.CreateCachedContentConfig(
        display_name="checkmyc",
        system_instruction=sys_prompt,
        contents=[
            types.Content(
                role="user",
                parts=[types.Part.from_text(text=
                    static_context)],
            )
        ],
        ttl="300s",
    ),
)
```

By abstracting these vendor-specific caching behaviors, *checkmyc* ensures a balance between high-performance execution and economic sustainability without compromising the deterministic nature of the experimental setup.

4.8 Evaluation Models and Output Generation

The framework’s evaluation process is governed by a hierarchical *bottom-up* aggregation logic. Every qualitative evidence detected by the LLMs and every deterministic result from functional tests is mathematically integrated into a final grade through three levels of aggregation, controlled by the weights defined in the configuration files. The resulting scores, together with the metadata and weighting parameters, are then organized to generate the comprehensive `checkmyc` output report.

4.8.1 Level 1: Atomic Evaluations (Per-Topic and Per-Test)

The first level computes scores for the elementary units of assessment: qualitative topics and deterministic tests.

Qualitative Topic Score (S_{T_i})

To formalize the topic score S_{T_i} , we define $|\overline{C_i^{g,c}}|$ as the number of conditions of type (g, c) effectively triggered by the LLM for topic i . Furthermore, we define an activation factor $\alpha_i \in \{0, 1\}$ for decisive conditions:

$$\alpha_i = \begin{cases} 0 & \text{if } \exists \text{ evidence} \in \{C_{i,j}^{-,decisive}\} \\ 1 & \text{otherwise} \end{cases} \quad (4.3)$$

The final topic score is calculated as:

$$S_{T_i} = \alpha_i \cdot \text{clip} \left(\max_i + \sum_{\substack{g \in \{+, -\} \\ c \in \{l, m, h\}}} |\overline{C_i^{g,c}}| \cdot w_i^{g,c}, 0, \max_i \right) \quad (4.4)$$

where `clip` guarantees S_{T_i} to be between 0 and $\max_i$.

Functional Test Scores (S_q)

In parallel, the system executes a set of deterministic tests $Q = \{gcc, time, pvcheck\}$. While these tests are not the primary object of the research, they are included to provide a holistic evaluation of the program, effectively anchoring the qualitative LLM feedback to objective functional metrics. Each test $q \in Q$ produces a score $S_q \in [0, 10]$ based on the following formalized logic:

- **Compiler Diagnostics (S_{gcc}):** The score is derived from the number of warnings generated by the `gcc` compiler using `-Wall` and `-Wextra` flags. If

the program fails to compile, the score is 0. Otherwise, it is calculated as:

$$S_{gcc} = \max(0, 10 - N_{warnings}) \quad (4.5)$$

where $N_{warnings}$ represents the total count of “warning”.

- **Execution Performance (S_{time}):** The score is determined by the elapsed execution time (t_{exec}) of the compiled program. The framework applies a discrete step function to assign a grade based on efficiency thresholds:

$$S_{time} = \begin{cases} 10 & \text{if } t_{exec} < 1.0\text{s} \\ 8 & \text{if } 1.0\text{s} \leq t_{exec} < 2.0\text{s} \\ 6 & \text{if } t_{exec} \geq 2.0\text{s} \\ 0 & \text{if the program crashes or times out} \end{cases} \quad (4.6)$$

- **Functional Correctness (S_{pcheck}):** This score represents the weighted average of the K functional test cases defined in `pvcheck.test`. The framework extracts raw percentage scores $s_k \in [0, 100]$ from the `pvcheck` CSV output and scales them to the framework’s decimal range using weights w_k specified in `questions.toml`:

$$S_{pcheck} = \frac{\sum_{k \in K} \frac{s_k}{10} \cdot w_k}{\sum_{k \in K} w_k} \quad (4.7)$$

4.8.2 Level 2: Qualitative and Functional Aggregation

The second level merges atomic results into two global macro-indicators.

Global Qualitative Score (S_{LLM})

Individual topic scores S_{T_i} are combined using weights W_{T_i} established in `llm.toml` (weight field):

$$S_{LLM} = \frac{\sum_{i=1}^n (S_{T_i} \cdot W_{T_i})}{\sum_{i=1}^n W_{T_i}} \quad (4.8)$$

Global Functional Tests Score (S_{Tests})

Similarly, deterministic test scores S_q are aggregated using weights W_q from the configuration file `questions.toml`:

$$S_{Tests} = \frac{\sum_{q \in Q} (S_q \cdot W_q)}{\sum_{q \in Q} W_q} \quad (4.9)$$

4.8.3 Level 3: Final Grade Synthesis (S_{Final})

The final stage synthesizes both assessments into a single score S_{Final} using the global weights w_{LLM} and w_{Tests} from `config.toml`:

$$S_{Final} = \frac{S_{LLM} \cdot w_{LLM} + S_{Tests} \cdot w_{Tests}}{w_{LLM} + w_{Tests}} \quad (4.10)$$

4.8.4 Artifact Materialization: JSON and HTML

The concluding step of the evaluation process is the materialization of results via the `save_json_and_html` utility. This mechanism ensures a strict separation between raw evaluation data and its visual presentation.

- **Structured JSON Output:** A comprehensive archive is saved, containing the student’s source code, the raw LLMresponse, normalized token usage, exact API call costs, and the full weight configuration. This file serves as the primary data source for the *Analysis Pipeline*.
- **Human-Readable HTML Report:** Utilizing the **Jinja2** engine and a dedicated `report_template.html`, the framework renders the JSON data into a dynamic visual report (see Figure 4.4).

This resulting interface is designed to maximize pedagogical impact through several interactive features:

- **Interactive Code Inspection:** The report displays the specific source code lines associated with each LLMevidence. These references are implemented as interactive triggers: when a user clicks on a line number or interval, a specialized `code-tooltip` appears, showing the relevant snippet of the student’s code with syntax-aware line numbering directly within the evaluation table.
- **Evidence Contextualization:** Each detected condition is identified by its `condition_id`. By interacting with these identifiers, users can reveal a tooltip containing the full description of the original pedagogical rule, allowing for an immediate comparison between the detected issue and the course requirements.
- **Visual Feedback System:** Qualitative assessments are enhanced by a color-coded signaling system (`status-circle`) where positive implementation choices (+), neutral informational triggers (=), and violations (−) are instantly recognizable.

CheckMyC Evaluation

Model: MODEL_NAME
Provider: PROVIDER_NAME
Final Score: 0.0
Call cost: \$0.0
Evaluated program: [example.c](#)

LLM Results

Name	LLM Score	Condition	Comments	Goodness	Criticality	Lines
Topic 1	0.0	#XX (?)	Generic comment.	⊖	medium	YY-ZZ
		#WW (?)	Generic comment.	⊕	high	TT
Topic 2	0.0	#PP (?)	Generic comment.	⊖	low	RR-VV

Practical Tips

LLM Scoring Logic

LLM FINAL SCORE - TOPIC WEIGHTS

Criteria	Weight
Modularity	0.0
Error handling	0.0

LLM TOPIC SCORE - COMMENT WEIGHTS

Topic	Criticality	Negative (-)	Positive (+)
Modularity	High	-0.0	+0.0
	Medium	-0.0	+0.0
	Low	-0.0	+0.0
Error handling	High	-0.0	+0.0
	Medium	-0.0	+0.0
	Low	-0.0	+0.0

Final Score - Split

FINAL SCORE SPLIT

Component	Weight	Score
LLM Evaluation (Qualitative)	0.0	0.0
Functional Tests (Quantitative)	0.0	0.0

Test Results & Weights

FUNCTIONAL TESTS - DETAILS

Test Type	Weight	Achieved Score
Warning	0.0	0.0
Performance	0.0	0.0
Pvcheck	0.0	0.0

PVCHECK SCORE - QUESTIONS WEIGHT

Q1: 0.0 Q2: 0.0 Q3: 0.0

Figure 4.4: Visual report of the *checkmyc* evaluation.

- **Scoring Transparency:** The report includes detailed breakdown tables for both LLM and functional scores, explicitly showing the weights and coefficients applied at each level of the bottom-up aggregation.

By automating this transition, *checkmyc* ensures that instructors can review high-level aggregates while students receive line-specific, actionable feedback grounded in the course’s pedagogical criteria.

4.9 Analysis Component

The *Analysis Component* is an independent module designed to provide the empirical evidence required to address the three Research Questions of this thesis. By decoupling the evaluation logic from the analytical tools, the framework ensures an impartial validation process, bridging the gap between technical implementation and

pedagogical objectives.

The analytical pipeline is organized into two primary phases, which provide the methodological foundation for the results discussed in the following chapter:

- **Phase I: Baseline and Calibration:** This stage establishes a *Human Gold Standard* through a dataset of 44 student programs. It defines the “Oracle performance” of the framework by calibrating the scoring weights, isolating the system’s intrinsic error from the LLMs detection capabilities.
- **Phase II: Performance Evaluation:** This phase benchmarks the LLMs across two analytical pillars: Accuracy (alignment with human scoring), and Stability and Reliability (consistency against model stochasticity).

The following subsections detail the the experimental setup employed to generate the comparative data and the statistical instruments.

4.9.1 Experimental Dataset and Human Gold Standard

The validation of the *checkmyc* framework is grounded in a dataset of 44 authentic student programs, derived from a single C programming exam assignment. This dataset reflects the diverse range of implementations, pedagogical nuances, and common coding errors encountered in an academic setting.

To provide a rigorous benchmark for the LLM-based evaluation, the entire dataset ($N = 44$) has been structured into a comprehensive *Human Gold Standard*, integrating two layers of validated information:

- **Numerical Gold Scores:** Every program is associated with per-topic numerical scores assigned by the course instructor, representing their holistic expert judgment.
- **Gold Evidences:** For all 44 programs, a complete set of ground-truth evidences has been manually produced, identifying specific `condition_id` triggers and corresponding code line references.

Bridging LLM Reasoning and Human Judgment

The dual nature of this dataset ($N = 44$) is fundamental to isolating the error base introduced by the evidence-based scoring mechanism. In the *checkmyc* framework, the extraction of specific evidences functions as a *bridge* between the raw LLM output and the instructor’s final grade. By applying the scoring logic directly to the *Human*

Gold Evidences, it is possible to determine the intrinsic error of the weighting model itself, separate from any inaccuracies in the LLM’s detection capabilities.

To ensure this bridge provides a reliable baseline for comparison between LLMs and humans, the evidence-to-score model undergoes a two-stage calibration process:

1. **Model Validation (LOOCV):** A LOOCV is performed on the available gold evidences to assess the generalizability and robustness of the scoring logic across different student implementations.
2. **Full Dataset Optimization:** A final optimization is conducted on the entire dataset to identify the optimal weights that minimize the Mean Absolute Error (MAE) against the instructor’s numerical scores.

This calibration ensures that the framework’s internal “ruler” is accurately aligned with the instructor’s pedagogical criteria, making the subsequent comparison between the LLM’s performance and the human gold standard statistically sound and theoretically consistent.

4.9.2 Weight Calibration and Per-Topic Optimization

A central aspect of the implementation involves the calibration of the percentage-based weighting scheme to isolate the error base introduced by the scoring mechanism. Since each assessment topic possesses a unique density of conditions and specific pedagogical relevance, the optimization is not performed globally but is specifically tailored for each topic i .

The *Grid Search* algorithm is employed to find the optimal weight vector $\mathbf{P}_i^*$ that minimizes the cost function J , defined as the MAE between the scores derived from the manual gold evidences and the instructor’s numerical gold scores:

$$\min_{\mathbf{P}_i \in \Theta} J(\mathbf{P}_i) = \frac{1}{N} \sum_{k=1}^N |S_{Final,k}(\mathbf{P}_i) - S_{Human,k}| \quad (4.11)$$

In this formulation, the parameters are strictly defined as follows:

- i : The index representing the specific **evaluation topic** (e.g., Modularity, Error Handling).
- N : The total number of programs in the experimental dataset ($N = 44$), for which both numerical scores and manual evidences are available.
- k : The index of the individual **student program** within the set.

- $S_{Human,k}$: The reference **gold score** assigned by the instructor to program k for topic i .
- $\mathbf{P}_i$: The vector of percentage weights for topic i , representing the impact of each criticality level c and goodness g :

$$\mathbf{P}_i = [P_i^{+,low}, P_i^{+,med}, P_i^{+,high}, P_i^{-,low}, P_i^{-,med}]$$

- $S_{Final,k}(\mathbf{P}_i)$: The score calculated by the framework for program k . During the LOOCV process, this score is computed using weights $\mathbf{P}_i$ derived from the remaining $N-1$ programs to ensure an unbiased estimation of the generalization error.

The optimization logic operates within a constrained parameter space Θ , explored through a discrete grid with a **step of 5%**. The search ensures that $\sum_c P_i^{g,c} = 100\%$ for each goodness category $g \in \{+, -\}$, subject to a strict hierarchical constraint ($P_{g,high} \geq P_{g,med} \geq P_{g,low}$) to ensure pedagogical consistency.

Model Validation Analysis

The effectiveness of the calibration was evaluated using the LOOCV methodology on the full dataset ($N = 44$). This approach provides a robust estimate of the framework’s generalization capability based on the available dataset. The results, summarized in Table 4.3, compare the LOOCV MAE against a *Heuristic MAE*, which was calculated using the weighting weights formally employed by the instructor during the manual grading process.

For the majority of the topics, the optimization process successfully identified weighting patterns that reflect the instructor’s judgment more accurately than the baseline heuristic. The most significant improvement is observed in *Code generality and quality*, where the error reduction reaches 51.70% (from 2.977 to 1.438). This result indicates a high degree of pedagogical consistency across the dataset for this topic: the model is able to learn robust weights from the 43 training samples that generalize effectively to the excluded test sample, successfully overcoming the “generalization penalty” inherent to cross-validation. Positive reductions were also achieved for Correct use of dynamic memory (12.24%) and Error handling (4.79%), confirming the framework’s ability to refine the instructor’s initial weight distribution.

Conversely, for the topics Modularity and Appropriate data structures, the LOOCV MAE was higher than the heuristic one, resulting in a deterioration (negative

Table 4.3: MAE computed on the full dataset ($N = 44$) using LOOCV to estimate generalization performance.

Topic	Heuristic	LOOCV	Reduction %
Modularity	0.600	0.711	-18.50%
Correct use of dynamic memory	1.193	1.047	12.24%
Appropriate data structures	1.100	1.733	-57.55%
Error handling	1.588	1.512	4.79%
Code generality and quality	2.977	1.438	51.70%

reduction percentage). In the first topic, the error increased by 18.50% (from 0.600 to 0.711), while in the other one is 57.55% higher than the heuristic baseline (from 1.100 to 1.733). This suggests the limitation of the available dataset, resulting in a higher sensitivity of these two topic to specific outliers, where the removal of a single sample during training significantly shifts the optimal weights. In such cases, the heuristic weights remain more stable than the calibrated ones.

Full Dataset Optimization Analysis

The final optimal weights $\mathbf{P}_i^*$ used for the comparative analysis between the LLM and the instructor are obtained through a final optimization pass on the entire dataset of 44 programs. The resulting *Final Optimized MAE* values represent the lower bound of the error (Oracle performance) for the proposed framework; as such, they serve as the definitive baseline for benchmarking the LLM’s evaluative capabilities. These results, alongside the corresponding Bias and Spearman Correlation metrics, are detailed in Table 4.4.

The analysis of the optimized MAE reveals that the error remains consistently below the 1.4 threshold across all evaluation topics. Notably, the Modularity topic achieved the highest precision, yielding a minimum MAE of 0.533. This suggests that the framework, when perfectly calibrated, can approximate human judgment with significant accuracy.

Analysing the Bias values we notice a general tendency of the framework to slightly overestimate the scores compared to the instructor’s judgment, as evidenced by the positive values across four out of five topics. The highest bias (0.628) is observed in Error handling, while a notable exception occurs in Appropriate data structures, where the negative bias (-0.733) reveals a systematic underestimation. This suggests that for this specific topic, the weighted impact of detected errors

Table 4.4: Final Optimized Metrics for each topic after calibration on the full dataset.

Topic	MAE	Bias	Spearman
Modularity	0.533	0.333	0.555
Correct use of dynamic memory	1.017	0.500	0.765
Appropriate data structures	1.067	-0.733	0.739
Error handling	1.356	0.628	0.878
Code generality and quality	1.400	0.533	0.766

might be more penalizing than the instructor’s subjective perception.

Regarding the Spearman Correlation, the results demonstrate a strong monotonic relationship between the framework’s outputs and the human gold standard. The Error handling topic stands out with an exceptionally high correlation ($\rho = 0.878$), indicating that the model is highly effective at preserving the relative ranking of students even when the absolute error (MAE) is higher. Conversely, while Modularity achieved the lowest MAE, it also displays the lowest Spearman correlation (0.555). This discrepancy suggests that while the absolute scores for Modularity are numerically close to the gold standard, the model finds it more challenging to consistently rank students’ performances in this category, likely due to a lower variance in the instructor’s grades or a more discrete distribution of the pedagogical criteria involved.

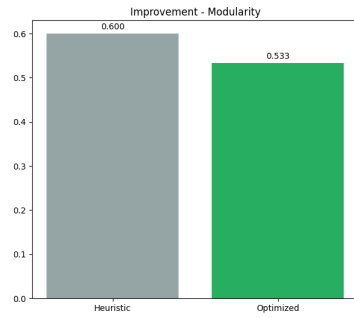
Overall, the combination of high Spearman coefficients (averaging above 0.74 for the remaining topics) and controlled MAE confirms that the optimized weighting scheme provides a solid foundation for the subsequent LLM evaluation phase.

Figure 4.5 and Figure 4.6 illustrate the error reduction (left) and the optimization convergence path (right) for each topic. The convergence plots show the algorithm’s stable descent toward the global minimum, confirming that the 5% grid step is adequate to capture the evaluative nuances across the entire experimental population.

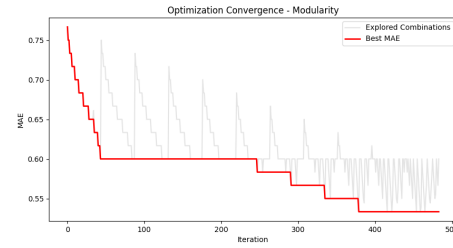
4.9.3 Metrics for Scoring and Detection Accuracy

The analysis pipeline implements several statistical metrics to quantify the alignment between LLMs and human graders:

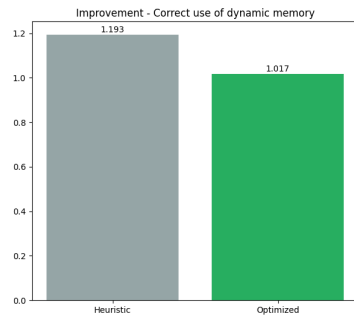
- **Scoring Accuracy:** Evaluated through MAE. Let y_i be the human score and



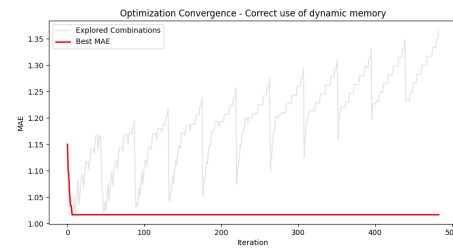
(a) MAE Comparison: Modularity



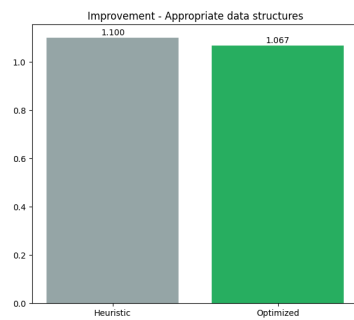
(b) Convergence: Modularity



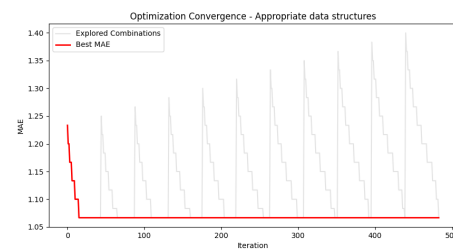
(c) MAE Comparison: Memory



(d) Convergence: Memory

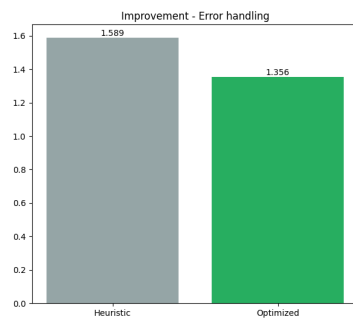


(e) MAE Comparison: Data Structures

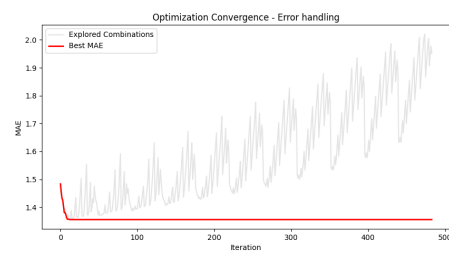


(f) Convergence: Data Structures

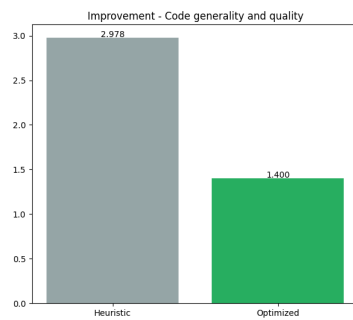
Figure 4.5: Optimization results (Comparison and Convergence) for the first three topics.



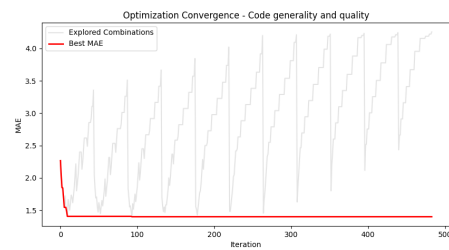
(a) MAE Comparison: Error Handling



(b) Convergence: Error Handling



(c) MAE Comparison: Code Quality



(d) Convergence: Code Quality

Figure 4.6: Optimization results for Error Handling and Code Quality.

$\hat{y}_i$ the predicted score for program i :

$$MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i| \quad (4.12)$$

MAE provides a direct measure of the average error magnitude.

- **Monotonicity and Rank:** The *Spearman Rank Correlation* (ρ) assesses the framework’s ability to preserve the relative ranking of students:

$$\rho = 1 - \frac{6 \sum d_i^2}{N(N^2 - 1)} \quad (4.13)$$

where d_i is the difference between the human gold scores and model scores.

- **Evidence Detection:** Precision (P), Recall (R), and F1-score are computed to evaluate the accuracy of condition triggering:

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN}, \quad F1 = 2 \cdot \frac{P \cdot R}{P + R} \quad (4.14)$$

where:

- **True Positives (TP):** number of correctly identified evaluation conditions present in the script by the LLM;
- **False Positives (FP):** number of conditions incorrectly flagged by the LLM that are not present in the gold evidences;
- **False Negatives (FN):** number of conditions that the LLM failed to detect.

These are calculated for both “ID-only” matches (conceptual alignment) and “Full-matches” (spatial localization on specific code lines).

4.9.4 Stability and Reliability Analysis

Since the final scores are a direct numerical derivation of the qualitative evidence identified by the models, the stability analysis follows a bottom-up approach, starting from the consistency of the extracted evidence sets and moving to the reproducibility of the assigned grades.

The stability of the qualitative feedback is quantified by measuring the consistency of the evidence sets detected across the different stochastic iterations. This is achieved through the *Jaccard Similarity* coefficient (J), calculated for each pair of runs as the

ratio of the size of the intersection to the size of the union of the detected condition sets:

$$J(E_a, E_b) = \frac{|E_a \cap E_b|}{|E_a \cup E_b|} \quad (4.15)$$

where E_a and E_b represent the sets of `condition_id` triggered in run a and run b , respectively. A Jaccard index of 1 indicates perfect identity between the qualitative feedbacks, whereas 0 denotes complete divergence.

Subsequently, systemic reliability is assessed by evaluating how this qualitative consistency translates into stable scoring. This is measured using the ICC(1,k), which quantifies the consistency of scores across the k performed LLM runs:

$$ICC(1, k) = \frac{MS_{between} - MS_{error}}{MS_{between}} \quad (4.16)$$

where $MS_{between}$ and MS_{error} represent the mean squares between subjects and the residual error, respectively.

While the ICC(1,k) evaluates the reliability of the student ranking (relative stability), it does not account for the absolute magnitude of score fluctuations. To address this, the STD is employed to quantify the absolute precision of the model for each specific topic. For a given assignment i and topic j , the STD is calculated over the k runs as follows:

$$STD_{i,j} = \sqrt{\frac{1}{k-1} \sum_{r=1}^k (x_{i,j,r} - \bar{x}_{i,j})^2} \quad (4.17)$$

where $x_{i,j,r}$ represents the score assigned in run r , and $\bar{x}_{i,j}$ is the mean score across all runs for that specific topic. This integrated approach ensures that the ICC(1,k) validates the reproducibility of the relative classification, while the STD measures the degree of uncertainty associated with the individual grade assignment.

Chapter 5

Results

This chapter presents the experimental evaluation of the proposed framework, analyzing the performance of different LLMs in emulating human assessment for C programming assignments. A fundamental premise for interpreting the results is the concept of *Oracle Performance*, established in the previous optimization phase (Section 4.9.2) and represented by the *Final Optimized MAE*. In fact the LLM-based evaluation is strictly dependent on the underlying per-topic weighted model; the *Final Optimized MAE* obtained using manual gold evidences represents the **generalization structural limit** of the system. Consequently, the following analyses do not merely measure the absolute error of the LLMs, but rather the “technological gap” introduced by the automated extraction process.

The results are organized as follows:

- **Human-LLM Alignment:** A direct comparison between LLM-generated scores and human gold scores, complemented by an analysis of the accuracy in extracting code evidences.
- **Generative Stability:** An investigation into the consistency of the models across three independent runs, evaluating the variance in both scoring and evidence detection.
- **Cross-Model Comparison:** A study of the correlations and divergences between the different LLMs to determine their interchangeability and architectural biases.

5.1 Human-LLM Alignment Analysis

This section evaluates the alignment between the scores generated by the LLMs considered in this work and the human-gold standard. Given that the final assessment is strictly derived from the evidence extracted by the models, it is first necessary to evaluate the LLMs’ proficiency in identifying such evidence. By adopting this hierarchical approach, the subsequent score analysis will not be treated as an isolated numerical output, but rather as a direct consequence of the models’ evidentiary grounding. This allows us to determine whether a high MAE or a significant Bias is rooted in a detection failure during the evidence extraction phase (e.g., an omission bias leading to under-scoring) or in a logic error during the final score synthesis due to the used conditions’ weights.

5.1.1 LLM Evidence Extraction Capabilities

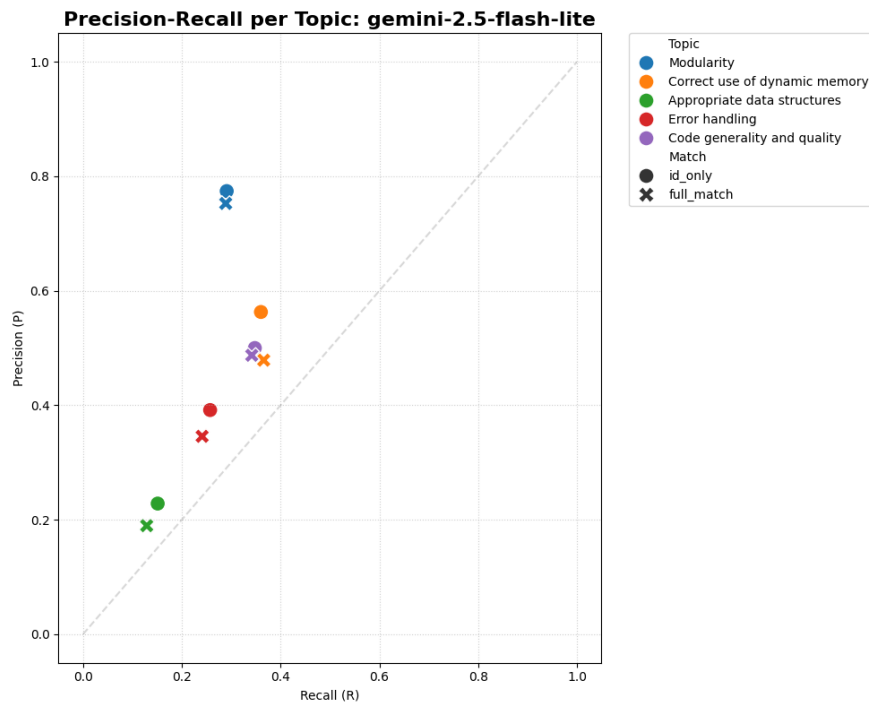
Two matching modalities are defined to verify the correspondence between the evidence extracted by the LLM (E_{LLM}) and the gold standard evidence (E_{Gold}):

- **ID-only Match:** A match is recorded based solely on the correspondence between the condition identifiers (`condition_id`). This metric evaluates the model’s ability to categorize technical patterns correctly, regardless of their position in the source code.
- **Full-match:** This criterion requires both `condition_id` correspondence and a spatial overlap between the code lines (`lines`) referenced by the condition. Let L_{Gold} be the set of lines identified in the gold standard and L_{LLM} the set of lines specified by the model; a full-match is satisfied if and only if:

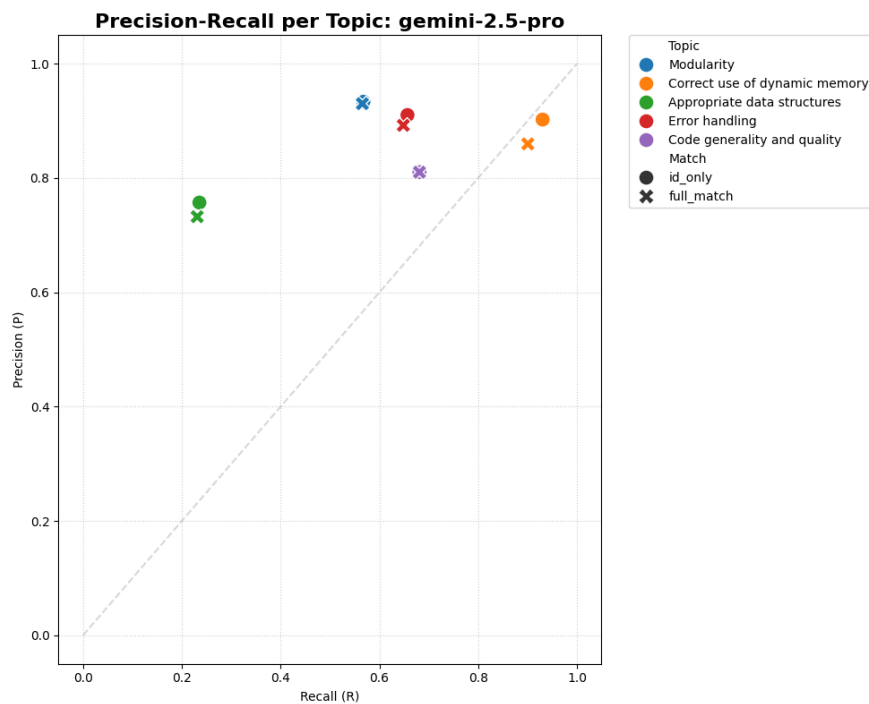
$$L_{Gold} \cap L_{LLM} \neq \emptyset \quad (5.1)$$

The performance of these matching modalities is visualized for each model and topic in Figures 5.1 and 5.2. These scatter plots represent Recall on the x-axis and Precision on the y-axis. Based on the distribution of points, we can characterize the LLMs’ behavior across four distinct functional regions:

- **Upper-Left (High Precision, Low Recall):** The model is *reliable but incomplete*. It rarely generates false positives (hallucinations), yet it suffers from an omission bias, failing to capture the full spectrum of valid conditions.

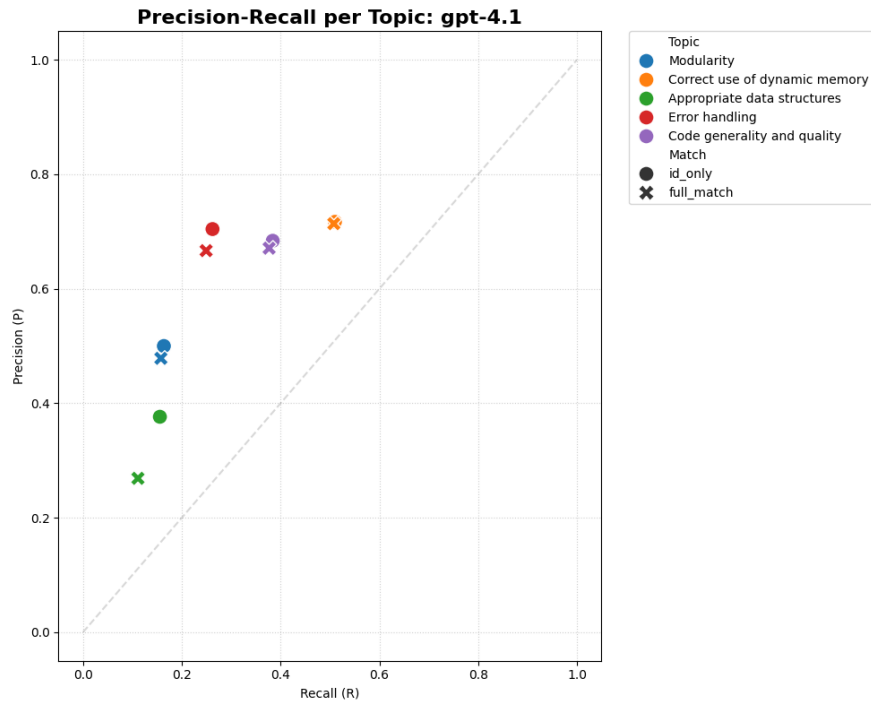


(a) Gemini 2.5 Flash Lite

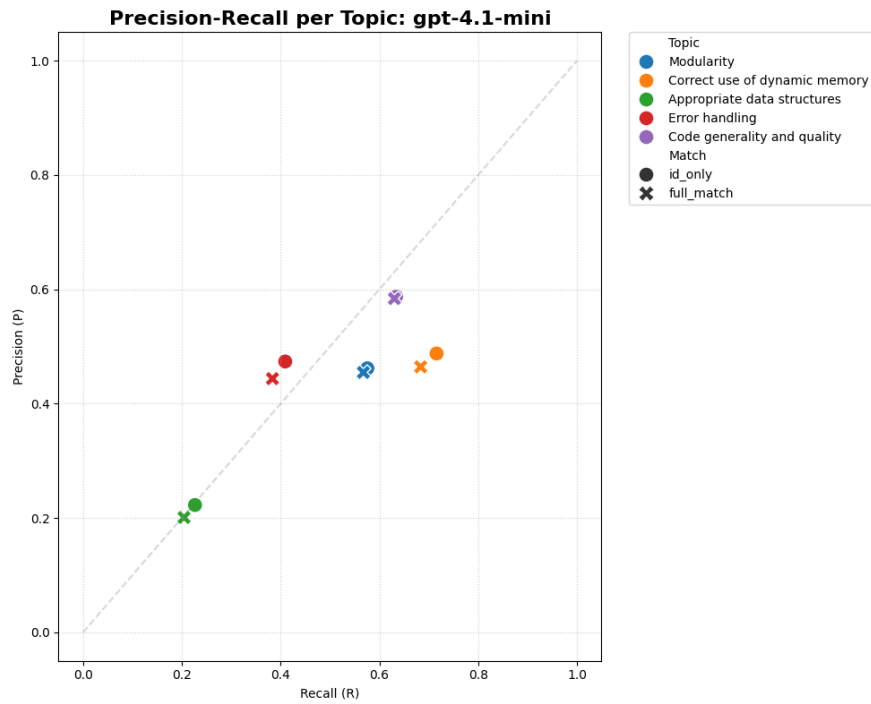


(b) Gemini 2.5 Pro

Figure 5.1: Precision-Recall scatter plots of the examined Gemini LLMs.



(a) GPT 4.1



(b) GPT 4.1 Mini

Figure 5.2: Precision-Recall scatter plots of the examined GPT LLMs.

- **Upper-Right (High Precision, High Recall):** The model is both *reliable and exhaustive*. It successfully identifies the majority of correct conditions while maintaining a low rate of extraneous or incorrect detections.
- **Lower-Left (Low Precision, Low Recall):** The model exhibits *poor performance*. It is characterized by low sensitivity and a high propensity for hallucinations, failing to grasp the underlying logic of the evaluation topic.
- **Lower-Right (Low Precision, High Recall):** The model is *exhaustive but noisy*. While it captures most of the ground-truth conditions, it tends to over-report evidence, frequently identifying benign code patterns as evaluative flaws.

According to this taxonomical subdivision and the average precision and recall for both matching modalities across all models (shown in Table 5.1), the considered LLMs can be categorized as follows:

- **Gemini 2.5 Pro:** This model represents the state-of-the-art for evidence extraction within our framework. With an average precision exceeding 0.85 and a recall consistently above 0.60, it frequently occupies the upper-right region. Notably, in the Correct use of dynamic memory topic, it achieves near-perfect alignment ($P \approx 0.90$, $R \approx 0.93$). However, in technically dense areas like Appropriate data structures, it moves toward the upper-left quadrant, becoming more conservative.
- **GPT-4.1:** This model exhibits a relative high degree of reliability (avg. Precision ≈ 0.60) but a significantly lower sensitivity compared to Gemini 2.5 Pro (avg. Recall ≈ 0.29). It is quite effective at avoiding hallucinations but suffers from a marked omission bias, identifying only a subset of the human-gold conditions. This indicates that while the model is acceptably trustworthy when it provides feedback, it lacks the exhaustiveness required for a complete automated assessment.
- **GPT-4.1-mini:** In contrast to its larger counterpart, the mini version shows an increased recall (avg. $R \approx 0.51$) at the expense of precision (avg. $P \approx 0.44$). It captures more conditions than GPT-4.1 but introduces a higher volume of false positives. This behavior suggests a tendency to over-report evidence, making it less reliable for direct student feedback without further filtering.

Table 5.1: Average Evidence Extraction Metrics per Model.

Model	Match Type	P	R	F1-Score
Gemini 2.5 Pro	ID-only	0.86	0.61	0.72
	Full-match	0.85	0.61	0.71
GPT-4.1	ID-only	0.60	0.30	0.40
	Full-match	0.56	0.28	0.38
GPT-4.1-mini	ID-only	0.45	0.51	0.48
	Full-match	0.43	0.49	0.46
Gemini 2.5 Flash-lite	ID-only	0.49	0.28	0.36
	Full-match	0.45	0.27	0.34

- **Gemini 2.5 Flash-lite:** This model demonstrates the lowest alignment with human-gold evidence. With average precision below 0.50 and recall near 0.28, it frequently identifies incorrect patterns while missing the majority of the ground truth. Its performance indicates that the “lite” architecture lacks the reasoning depth necessary to navigate the complexities of C programming rubrics through zero-shot prompting.

The hallucination rates for each LLM across the evaluated topics are illustrated in Figure 5.3. This metric, defined in Equation 5.2, represents the percentage of predicted evidences that are false positives, providing a measure of the model’s tendency to detect incorrect conditions:

$$H_{Model} = \frac{FP}{TP + FP} \cdot 100 \quad (5.2)$$

These results further identify Gemini 2.5 Pro as the most robust model regarding evidence extraction accuracy. Across all models, the topic Appropriate data structures exhibits the highest incidence of hallucinations. However, a significant performance gap is observed: while Gemini 2.5 Pro maintains an acceptable hallucination rate below 30%, the other three models exceed 70%, effectively rendering them unreliable for this specific category.

5.1.2 LLM Score Alignment Analysis

As anticipated in Section 4.9.2, the evaluation of scoring performance must be contextualized within the constraints of the Oracle performance, which represents

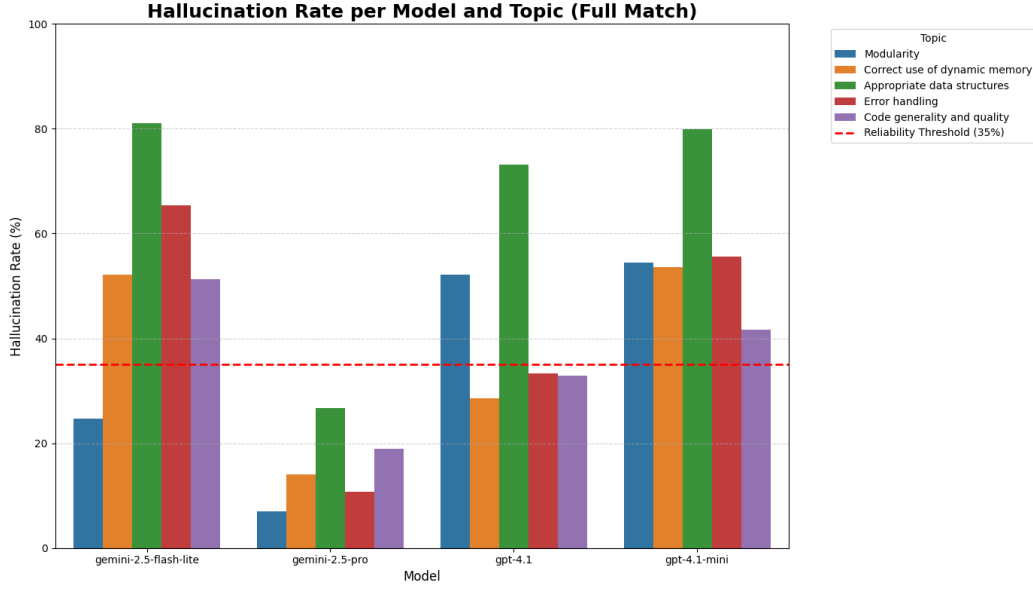


Figure 5.3: Hallucination rate of LLMs across topics

the intrinsic ceiling of the evidence extraction process. Consequently, the three metrics utilized in this analysis (MAE, Bias, and the Spearman Correlation Coefficient) are presented alongside their respective *Technological Gaps* in Table 5.2.

Moreover to ensure the integrity of the subsequent evaluation, a reliability threshold of 35% has been established. For this reason GPT-4.1-mini is excluded from the following analysis. The subset of model-topic pairs that strictly satisfy this reliability constraint and are therefore included in the subsequent score alignment analysis comprises:

- **Gemini 2.5 Pro:** all evaluated topics;
- **Gemini 2.5 Flash-Lite:** limited to the Modularity topic;
- **GPT-4.1:** Correct use of dynamic memory, Error handling, and Code generality and quality.

The reported gaps for the defined models subset, calculated relatively to the reference values established in Table 4.4, quantify the actual divergence between the LLM and human judgment. Table 5.3 report the gap value interpretation basing on its sign for each scoring metric.

The visual representation of these gaps across all topics and models is provided in Figures 5.4, 5.5, and 5.6.

Table 5.2: Detailed Scoring Metrics and Tech Gaps per Topic and Model.

Model	Topic	MAE Gap	Bias Gap	Sp. Gap
Gemini 2.5 Pro	Modularity	0.84 0.31	0.13 -0.20	0.43 -0.13
	Dynamic Memory	1.30 0.29	-0.08 -0.58	0.66 -0.11
	Data Structures	1.24 0.18	-0.34 0.39	0.69 -0.04
	Error Handling	1.78 0.43	1.10 0.50	0.80 -0.07
	Code Quality	2.52 1.12	1.5 0.97	0.17 -0.60
Gemini 2.5 Flash-lite	Modularity	1.62 1.09	-0.67 -1.00	0.35 -0.21
GPT-4.1	Dynamic Memory	1.58 0.56	1.49 0.99	0.72 -0.04
	Error Handling	3.80 2.44	3.31 2.68	0.51 -0.37
	Code Quality	4.39 2.99	4.37 3.83	0.27 -0.50

Table 5.3: Interpretation of Technological Gaps for Scoring Metrics.

Metric	Positive Gap (+)	Negative Gap (-)
MAE	Lower Accuracy: The model has a higher average error magnitude than the baseline.	<i>(Theoretically unlikely):</i> The model’s error is lower than the baseline.
Bias	Over-scoring: The model is more lenient or “hallucinates” merits, assigning higher scores than the human/oracle.	Under-scoring: The model is more severe or “omits” valid patterns (Omission Bias), assigning lower scores.
Spearman	<i>(Theoretically unlikely):</i> The model ranks assignments better than the baseline.	Lower Ranking Consistency: The model is less capable of ordering students by quality compared to the human/oracle.

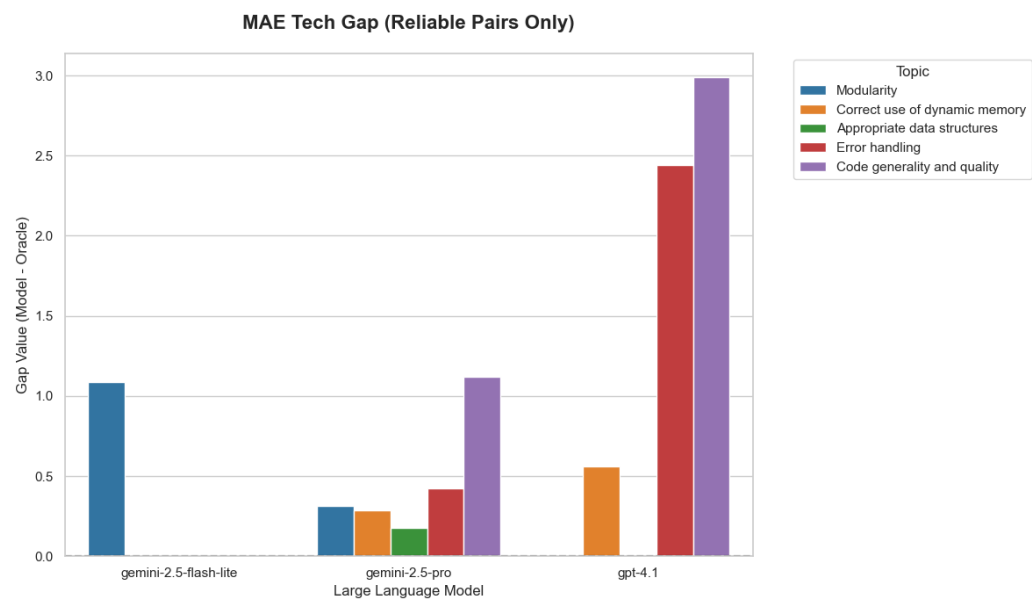


Figure 5.4: MAE Gap for reliable models across topics.

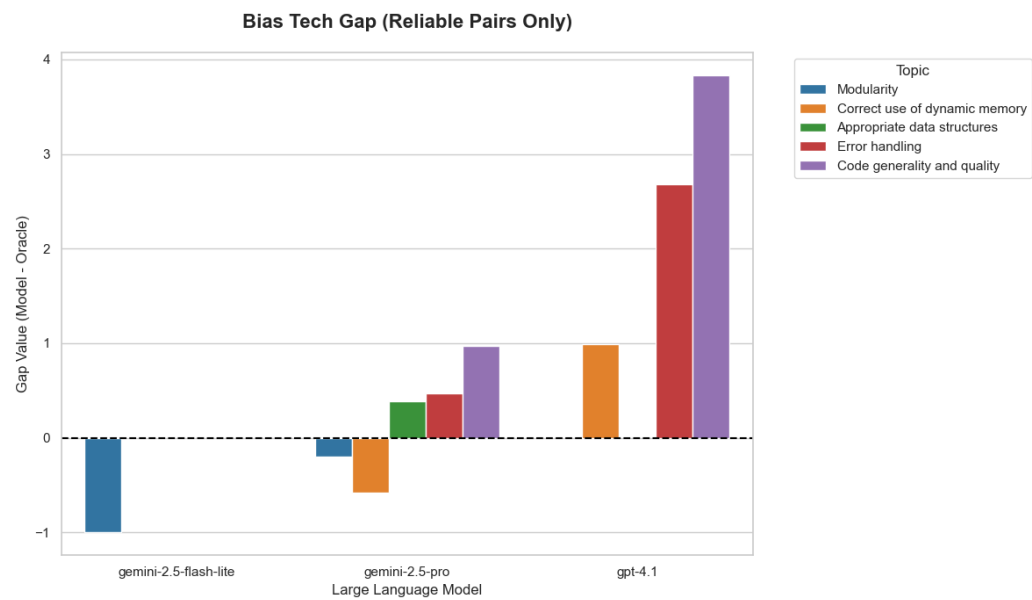


Figure 5.5: Bias Gap for reliable models across topics.

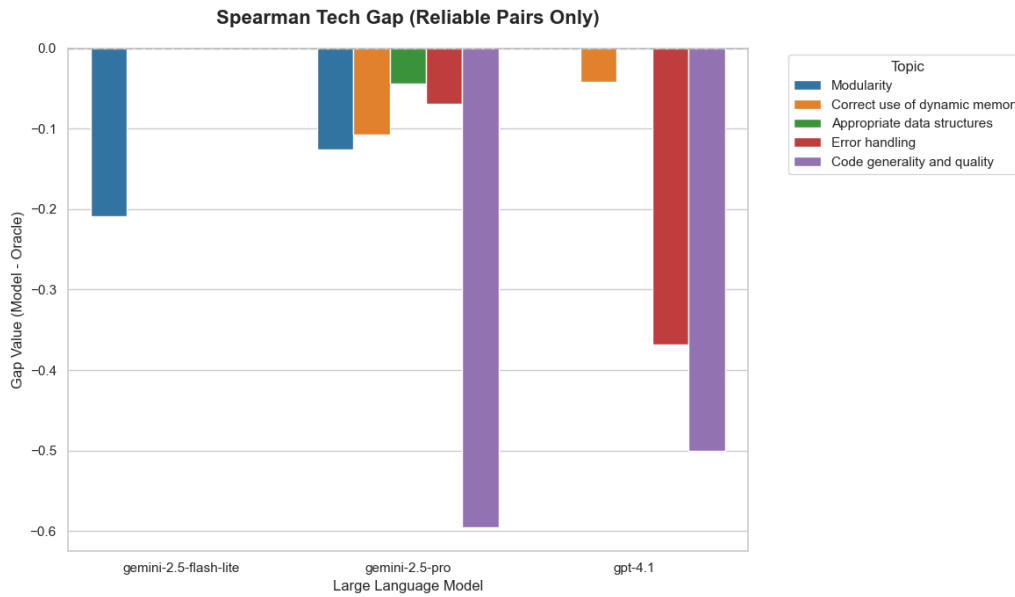


Figure 5.6: Spearman Coefficient Gap for reliable models across topics.

By cross-referencing these scoring metrics with the behavioral analysis performed in the previous section, we can derive a comprehensive understanding of the models' evaluative logic:

- Gemini 2.5 Pro:** As the state-of-the-art model in our framework, Gemini 2.5 Pro exhibits the highest degree of alignment with human scoring. As shown in Figure 5.4, this model consistently maintains the lowest MAE gaps across all topics. This result directly correlates with its superior evidence extraction performance ($F1 = 0.72$), demonstrating that a near-perfect identification of code patterns is a prerequisite for accurate grade assignment. The Bias Gap remains consistently lower than 1.0, with its maximum value registered for Code generality and quality, reflecting the previous consideration about the MAE Gap for the same topic. Its ranking consistency, indicated by the Spearman gap in Figure 5.6, remains the most stable (except for Code generality and quality), confirming its ability to correctly differentiate student performance levels across most of the topics. Overall, the model's performance aligns closely with the Oracle, with the notable exception of the Code generality and quality topic, where a deviation is observed.
- GPT-4.1:** The scoring metrics for GPT-4.1 reflect its good precision coupled with low sensitivity. In Figure 5.5, all the three considered topics show a positive Bias Gap, particularly pronounced in Error handling and Code generality and

quality. This systemic over-scoring is a direct consequence of the model’s “omission bias”: because it identifies only a subset of the gold conditions, it fails to detect negative conditions triggered in the code. This leads to a higher MAE gap compared to Gemini 2.5 Pro (and, consequently, to the Oracle), and to lower the ranking consistency, suggesting that GPT-4.1 is not capable of ordering students by quality except for Correct use of dynamic memory. From these analysis, this model proves to be acceptable only for Modularity evaluations.

- **Gemini 2.5 Flash-lite:** For Gemini 2.5 Flash-lite, only the Modularity topic met the reliability threshold. Despite passing the hallucination filter for this specific topic, it shows significant performance degradation compared to the larger architectures. The model exhibits notable gaps values for all the three metrics, suggesting a reduced ability to correctly evaluate this topic. Even when the model identifies correct patterns, its limited reasoning depth, indicated by the lower alignment metrics, prevents it from translating those evidences into a nuanced and accurate final score. This confirms that the “lite” architecture struggle to navigate the complexities of the C programming topics.

5.2 Stability and Reliability Results

The stability of the framework was evaluated by analyzing the consistency of the qualitative evidence extraction and the reproducibility of the quantitative scoring across three stochastic LLM runs. The results, subdivided by model and topic, are summarized in the heatmaps for Jaccard Similarity (Figure 5.7), ICC(1,k) (Figure 5.8), and Score STD (Figure 5.9).

5.2.1 Qualitative Evidence Stability

The consistency of the identified evidence sets, measured via the Jaccard Similarity index, provides insights into the models’ deterministic behavior at the prompt-level. As illustrated in Figure 5.7, Gemini 2.5 Pro demonstrates exceptional qualitative stability, with Jaccard values consistently exceeding 0.80. Notably, the Correct use of dynamic memory topic achieves a stability index of 0.89, indicating that the model triggers nearly identical sets of conditions across different iterations. GPT-4.1 shows a good degree of stability with a value around 0.80, except for Error handling topic, where the stability drops to 0.56. Gemini 2.5 Flash-lite and GPT-4.1-mini exhibit the lowest qualitative stability, with values reaching their lowest values for

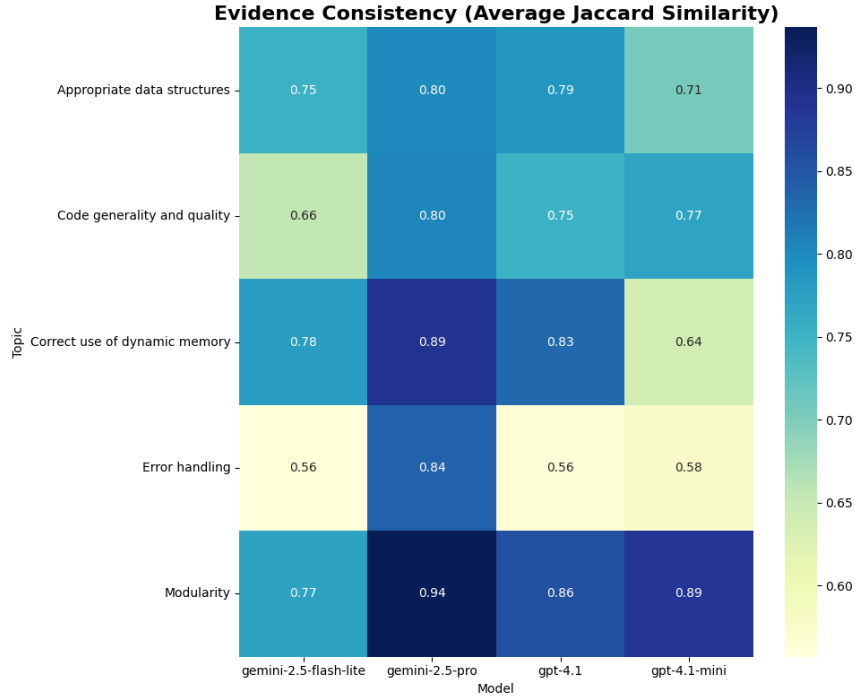


Figure 5.7: Jaccard index heatmap for models across all topics.

Error handling, confirming that smaller architectures struggle to maintain a coherent reasoning path in the defined topics.

5.2.2 Quantitative Score Reliability

The transition from qualitative evidence to numerical scores is analyzed through the ICC(1,k) and the STD.

Ranking Consistency (ICC(1,k))

Figure 5.8 confirms that the framework maintains high reliability in terms of student ranking. Both Gemini 2.5 Pro and GPT 4.1 achieve ICC(1,k) values above 0.90 for most topics, while the other two models struggle to maintain a high ICC(1,k) value among all topics. Modularity and Appropriate data structure are the topics for which all models present an high score ordering. Even when the qualitative evidence varies (as seen before for the smallest architectures), the resulting scores maintain a good level of consistency relatively to ordering of the assignments (never lower than 0.70).

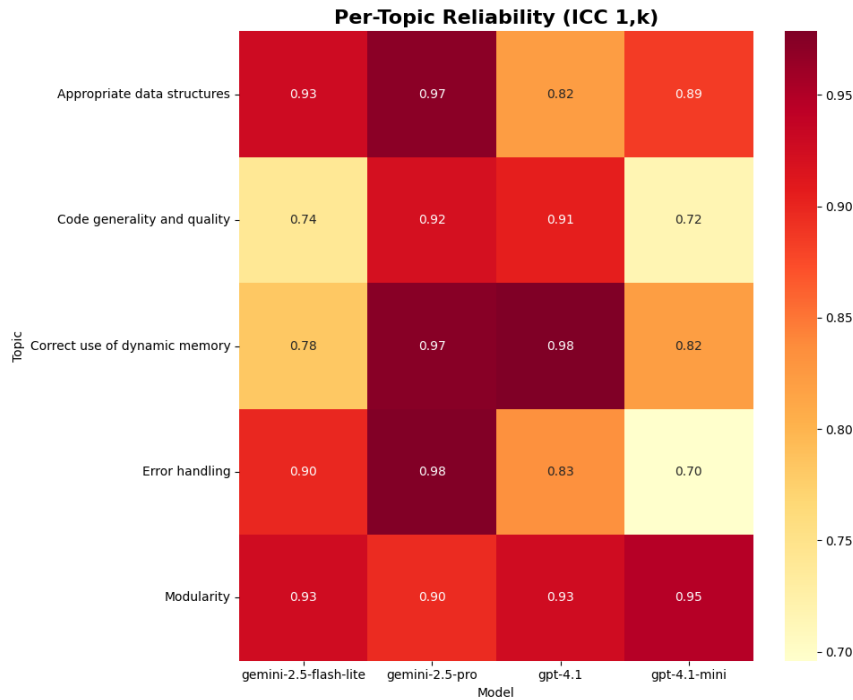


Figure 5.8: ICC(1,k) heatmap for models across all topics.

Score Precision (STD)

The absolute precision of the grading is visualized in the Standard Deviation heatmap (Figure 5.9). Gemini 2.5 Pro maintains a high level of precision, with an average STD below 0.57 across all topics. The most stable topic is Appropriate data structures ($STD = 0.29$), where the model’s conservative extraction profile leads to highly predictable scores. GPT-4.1 present even lower STD values compared to Gemini 2.5 Pro for Code generality and quality and Correct use of dynamic memory, but exceed the 1.0 value for the topics Appropriate data structure and Correct use of dynamic memory. Conversely, Gemini 2.5 Flash-lite and GPT-4.1-mini displays significant score dispersion, particularly in Correct use of dynamic memory with values near to 2.0. Such high fluctuations suggest that for models with smaller architectures, the grading process is heavily influenced by stochastic noise, making a single run insufficient for a fair automated assessment.

5.2.3 Summary of Stability Findings

The integrated analysis of these metrics highlights a fundamental correlation: higher reasoning depth (Gemini 2.5 Pro) results in superior Jaccard stability, which in turn minimizes score uncertainty (STD). While ICC(1,k) values suggest that all models

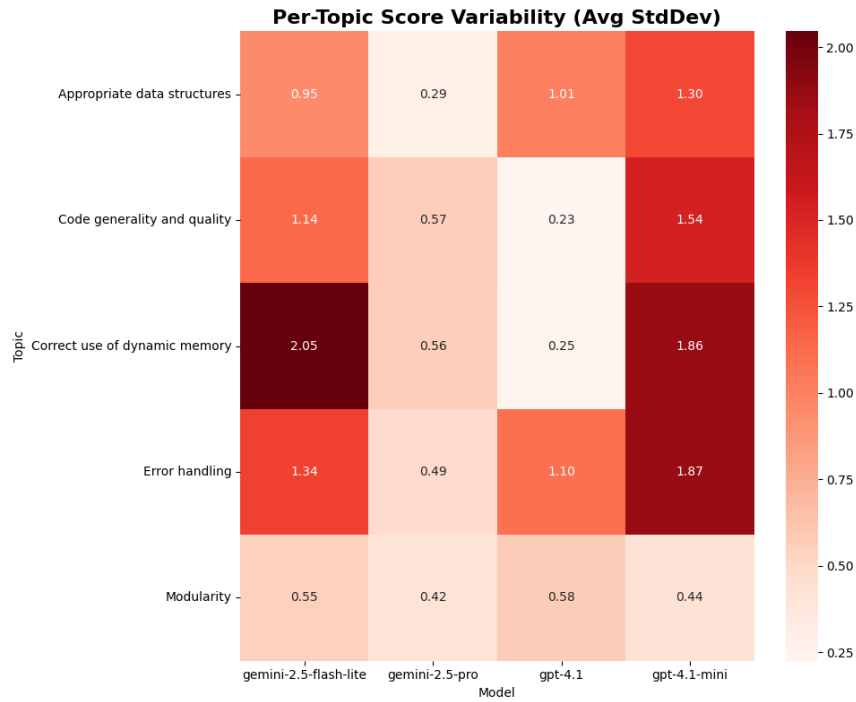


Figure 5.9: Standard Deviation heatmap for models across all topics.

can rank students with acceptable reliability, the high STD and lower Jaccard indices of the Flash-lite and GPT-4.1-mini models necessitate the use of multiple runs or more advanced sampling techniques to achieve the precision required for high-stakes academic grading. In the middle there's the GPT-4.1 model which offer a good level of stability only for the two topics Correct use of dynamic memory and Code generality and quality.

Chapter 6

Conclusions

This thesis investigated the viability of using Large Language Models to automate the assessment of C programming assignments by emulating human evaluative logic through prompt engineering and specific topic customization. Based on the experimental results, the following conclusions address the primary research questions of this study.

RQ1: Correspondence with Human Reference Evaluations The LLMs examined in this study exhibit divergent responses to this research question, identifying the underlying model architecture as a primary determinant of performance. Models characterized by smaller architectures, while cost-effective, demonstrate limited reasoning capabilities and are frequently compromised by hallucinations. Such behavior renders these models unsuitable for rigorous academic environments: although they may occasionally identify correct qualitative patterns within the source code, this detection remains partial and is often obscured by the presence of false positives, making it difficult to distinguish valid evidence from noise.

Conversely, this limitation is not universal across all models. One specific high-capacity model emerged for its distinctive performance which, while not infallible, positions it as a viable tool for student self-assessment and as a supportive asset for instructors. In an instructional context, even without achieving perfect alignment, this model can effectively identify programs affected by minor functional errors that nevertheless demonstrate a correct grasp and implementation of the course’s qualitative requirements.

RQ2: Stability and Internal Reliability Stability and internal reliability fluctuate significantly across different models and evaluation topics. Smaller architectures

exhibit intrinsic limitations in consistency, which precludes their safe and reliable application in academic grading. In contrast, larger models demonstrate high, and at times exceptional, stability across multiple iterations, thereby favoring their integration into automated assessment pipelines.

RQ3: Systematic Differences and Comparative Reliability As established in the preceding analysis and the results presented in Chapter 5, LLMs are not interchangeable for the evaluative tasks considered in this thesis. The determining factors for this lack of interchangeability include the depth of reasoning, directly correlated with the underlying architecture, the management of stochastic noise, the propensity for omission bias, and overall robustness against hallucinations.

6.1 Future Work

The findings of this research suggest several directions for future development:

- **Refinement of Prompt Engineering:** Investigating more sophisticated prompting techniques to further bridge the gap between automated and human assessment.
- **Evaluation of Next-Generation Models:** Extending the analysis to more advanced architectures beyond those considered in this study to determine if increased reasoning depth continues to yield proportional improvements in evaluative logic.
- **Cross-Paradigm Validation:** Extending the framework to object-oriented or functional programming languages to evaluate if the observed systematic differences between models remain consistent across different coding semantics.

Bibliography

- [1] Dang Anh-Hoang, Vu Tran, and Le-Minh Nguyen. Survey and analysis of hallucinations in large language models: attribution to prompting strategies or model behavior. *Frontiers in Artificial Intelligence*, 8:1622292, 2025.
- [2] Rishi Bommasani and et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021. <https://arxiv.org/abs/2108.07258>.
- [3] Mark Chen and et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. <https://arxiv.org/abs/2107.03374>.
- [4] Benjamin Simon Clegg. *The application of mutation testing to enhance the automated assessment of introductory programming assignments*. PhD thesis, University of Sheffield, 2021.
- [5] Serge Comb  fis. Automated code assessment for education: Review, classification and perspectives on techniques and tools. *Software*, 1(1):3–30, 2022.
- [6] Dante D Dixon and Frank C Worrell. Formative and summative assessment in the classroom. *Theory into practice*, 55(2):153–159, 2016.
- [7] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3):1–13, 2005.
- [8] Chongzhou Fang, Ning Miao, Shaurya Srivastav, et al. Large language models for code analysis: Do LLMs really do their job? In *USENIX Security Symposium*, 2024.
- [9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, and Daxin Jiang. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1536–1547, 2020.

- [10] V. Hackl, A. E. Müller, M. Granitzer, and M. Sailer. Is GPT-4 a reliable rater? evaluating consistency in GPT-4’s text ratings. *Frontiers in Education*, 8, 2023.
- [11] John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.
- [12] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.
- [13] Alon Jacovi and Yoav Goldberg. Towards faithfully interpretable NLP systems: How should we define and evaluate faithfulness? *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, pages 4198–4205, 2021.
- [14] Ziwei Ji et al. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.
- [15] Michał Jukiewicz. The future of grading programming assignments: The role of ChatGPT in automating assessment and feedback. *Thinking Skills and Creativity*, 52:101522, 2024.
- [16] Ayaan M. Kazerouni, James C. Davis, Arinjoy Basak, Clifford A. Shaffer, Francisco Servant, and Stephen H. Edwards. Fast and accurate incremental feedback for students’ software tests using selective mutation analysis. *Journal of Systems and Software*, 175:110905, 2021.
- [17] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19(1):1–43, 2018.
- [18] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.
- [19] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaoqing Shi. How consistent are humans when grading programming assignments? *ACM Trans. Comput. Educ.*, 25(4):1–37, 2025.

- [20] Michael Messer, Neil C. C. Brown, Michael Kölling, and Mingyu Shi. Machine learning-based automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 2023.
- [21] Maxwell I. Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, abs/2112.00114, 2021. <https://arxiv.org/abs/2112.00114>.
- [22] Ehud Reiter. A structured review of the validity of BLEU. *Computational Linguistics*, 44(3):393–401, 2018.
- [23] Arthur Rump, Vadim Zaytsev, and Angelika Mader. Requirements for an automated assessment tool for learning programming by doing. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 679–686. IEEE, 2025.
- [24] Rickard Stureborg, Dimitris Alikaniotis, and Yoshi Suhara. Large language models are inconsistent and biased evaluators. *CoRR*, abs/2405.01724, 2024. <https://doi.org/10.48550/arXiv.2405.01724>.
- [25] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves Chain of Thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
- [26] Sarah Wiegrefe and Yuval Pinter. Measuring association between labels and free-text rationales. *NAACL*, 2021.
- [27] Chengguan Xiang, Ying Wang, Qiyun Zhou, and Zhen Yu. Graph semantic similarity-based automatic assessment for programming exercises. *Scientific Reports*, 14:10530, 2024.
- [28] Wenjing Xie, Juxin Niu, Chun Jason Xue, and Nan Guan. Grade like a human: Rethinking automated assessment with large language models. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS 2025, Ho Chi Minh, Vietnam, November 16-19, 2025*, pages 21:1–21:8. ACM, 2025.

- [29] Mohamed Yousef, Karim Mohamed, Waleed Medhat, et al. BeGrading: large language models for enhanced feedback in programming education. *Neural Computing and Applications*, 37:1027–1040, 2025.