

Article

An LLM-Based Guided Programming Assistance System for Code Quality Feedback and Formative Assessment

Guoyang Liu 

School of Integrated Circuits, Shandong University, Jinan 250101, China; gyliu@sdu.edu.cn

Abstract

Recent advancements in Large Language Models (LLMs) hold significant promise for reshaping programming education. However, critical instructional challenges such as high failure and dropout rates, insufficient feedback, and inadequate support for students' independent analytical thinking remain prevalent. Addressing these gaps, this study introduces the Guided Programming and Analysis System (GPAS), an innovative educational approach leveraging LLM-based technology to enhance programming instruction. GPAS is designed to support autonomous learning processes by integrating multi-turn interactive thought guidance, code polishing, semantic annotation generation, and structured scoring mechanisms across multiple programming languages. Experimental results demonstrate significant effects were observed in correlation analysis with expert evaluations ($r = 0.668$, $p = 6.84 \times 10^{-12}$) and paired-sample tests on code and report-level improvements (effect sizes Cohen's $d = 0.92$ and 1.56 , respectively, $p = 3.56 \times 10^{-6}$ and $p = 1.64 \times 10^{-10}$). Additionally, findings revealed that the GPAS platform significantly improved code quality, particularly benefiting lower-achieving students, and effectively captured nuanced improvements in readability, structure, and boundary handling. Moreover, GPAS emphasizes the supportive role of educators, enabling them to focus more effectively on higher-order teaching tasks while the platform handles routine instructional feedback. Collectively, these results suggest that GPAS provides a promising framework for supporting learner-centered programming education and independent problem-solving processes.

Keywords: large language models; programming education; autonomous learning; GPAS platform; educational technology

1. Introduction

Learning to program is a challenging process that requires mastering logic, syntax, and problem-solving skills. High failure and dropout rates are common in introductory programming courses, often attributed to factors such as novice students' limited problem-solving skills, large class sizes that impede individualized feedback, and abstract concepts that beginners struggle to grasp [1,2]. In practice, the journey of learning to code frequently involves iterative trial-and-error, debugging, and overcoming syntax errors, which can be time-consuming and frustrating without proper guidance [1]. Consequently, there is a strong pedagogical imperative for intelligent feedback mechanisms that can provide immediate, tailored assistance to students as they work through coding problems. Timely, personalized feedback is essential for students learning programming [3], helping them correct misconceptions and maintain engagement.

Recent advancements in Large Language Models (LLMs), such as ChatGPT and DeepSeek, have significantly impacted programming education by virtue of their robust



Academic Editor: Jose María Alvarez Rodríguez

Received: 15 May 2026

Revised: 14 June 2026

Accepted: 18 June 2026

Published: 29 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

capabilities in natural language understanding and code generation [4]. Existing studies show that large language models, such as GPT-4, have achieved performance approaching or even surpassing expert-level accuracy on standard programming benchmarks such as HumanEval [5]. In higher education contexts, these models have been successfully integrated into various instructional scenarios, such as automated program repair, code annotation, and formative feedback generation, substantially enhancing instructional efficiency and enabling more personalized feedback mechanisms [6–8].

Predominantly, existing research emphasizes assessing LLMs' performance in completing programming tasks [9]. For instance, Savelka et al. [10] compared the effectiveness of GPT-3 and GPT-4 models on a set of 599 programming exercises derived from three Python courses, revealing significant improvements over model iterations. Initially, the GPT-3 model was unable to pass standard course assessments independently, whereas GPT-4 demonstrated the capability to successfully complete these tasks autonomously without human intervention. Additionally, leveraging LLMs for automated feedback generation has attracted considerable scholarly interest. Systems like CodeGrade employ LLM integration to automate grading processes and provide structured feedback [11]. However, most of these systems adopt a "student input, model output" interaction paradigm, which predominantly neglects students' underlying cognitive processes in programming. Such approaches risk diminishing students' conceptual understanding and deeper engagement with programming principles.

To mitigate the educational pitfalls associated with providing immediate solutions, researchers are increasingly advocating for pedagogically oriented interventions, integrating deliberate prompting and structured feedback. For example, Kargupta et al. [12] developed the TreeInstruct system, which leverages Socratic questioning techniques in a multi-turn interactive format. This strategy guides learners through a systematic debugging process, encouraging independent identification and rectification of programming errors and indicating substantial improvements in debugging efficacy and code quality compared to traditional, direct-answer paradigms. Consequently, educational-oriented prompting systems, which emphasize incremental guidance, multi-turn interactions, and constructive feedback loops, demonstrate greater alignment with pedagogical objectives. Such systems foster enhanced student comprehension, critical thinking, and progressive cultivation of programming skills, thereby exemplifying an effective integration of LLMs into educational practice.

The ability to independently write code remains a critical learning outcome in introductory programming courses. Traditionally, proficiency in coding skills is achieved through iterative practice on numerous small-scale coding exercises [13]. While contemporary LLMs are capable of solving introductory-level programming problems with high accuracy, thereby offering direct support to learners in need [14], there is growing concern regarding potential overreliance on automated code generation tools. Empirical evidence increasingly suggests that frequent utilization of LLMs by students can negatively impact their ability to independently decompose problems and engage in active problem-solving, creating a dependency that impedes programming proficiency when technological support is unavailable [15]. Kazemitabaar et al.'s empirical investigation into the Coding Steps platform further highlighted this issue, reporting that many learners tend to directly copy assignment prompts into models, subsequently adopting the generated code without meaningful engagement, modification, or comprehension [16]. To counteract these concerns, instructional systems must adopt interactive, structured pedagogical scaffolds, such as incremental prompting, visualization of code structures, and interactive feedback, to encourage deeper learner engagement and foster genuine understanding of programming logic. Additionally, implementing thoughtful constraints or "guardrail" strategies to dis-

courage direct, fully automated solutions is crucial [17]. These pedagogical interventions can effectively stimulate active learner participation in the problem-solving and debugging processes, ultimately facilitating sustained cognitive engagement and the independent development of programming skills.

Collectively, this work introduces a user-friendly LLM-based GPAS platform for programming education that integrates LLM-based support with a comprehensive, pedagogy-driven design. The goal of our GPAS platform is to provide students with rich, scaffolded feedback on their code assignments while employing strict guardrails to prevent dependency on direct answers. The key contributions of this paper are as follows:

- This study developed the GPAS platform, a novel LLM-based programming education platform integrating interactive scaffolding, code polishing, semantic annotation, and structured multi-dimensional scoring feedback, effectively addressing the shortcomings of existing platforms that lack continuous cognitive support and personalized guidance.
- Through rigorous experiments involving correlation analyses and paired-sample tests, the developed GPAS platform demonstrated strong reliability and sensitivity in scoring student submissions, especially in capturing improvements in code quality and programming proficiency following LLM-driven code polishing interventions.
- Compared to existing systems, the proposed GPAS platform uniquely supports multiple programming languages (including Python, C, MATLAB, Verilog, etc.) and emphasizes instructional guardrails that avoid direct code provision, fostering students' independent problem-solving, critical analysis skills, and promoting educational equity by significantly benefiting lower-performing students.

2. System Overview

Figure 1 illustrates the architecture of our Guided Programming Assistance System (GPAS), which integrates two key modules: the Assisted Programming Module and the Code Analysis Module. The system is designed as a closed-loop learning cycle—guidance, implementation, and feedback—centered on Student Input. Students can either submit programming problems (Problem Submission) or code (Code Submission). Upon problem submission, the Code Assistant provides Thought Guidance and Problem Analysis through Multi-turn Interaction, leading to a Final Output that reflects the student's reasoning process. For code input, the Code Analysis Module performs Code Analysis, Code Optimization, Code Refinement, and Semantic Annotation, culminating in Scoring Feedback. Both Final Output and Scoring Feedback can, based on student choices, be routed back to Student Input, helping students refine their understanding and form an iterative learning path. The platform leverages large language models (LLMs) behind a policy layer that restricts outputs to reasoning prompts, code refinement, semantic annotation, and rubric-based scoring with explanations, explicitly disallowing full solutions. Currently, GPAS supports MATLAB, Python, C, and Verilog and is tailored for lab-style programming education in engineering curricula. Two user interfaces expose the core functions:

- Guided Programming view (Figure 2): a multiturn dialogue workspace that elicits the learner's intent, decomposes tasks, and structures a solution path without emitting complete code.
- Code Analysis view (Figure 3): a code-centric workspace that accepts a learner's snippet and returns three independent artefacts, polishing suggestions, semantic annotations, and scoring feedback, under a fixed six-dimension rubric.

Both views share a consistent left–right layout: the left pane hosts inputs (problem text or code plus language selector); the right pane presents structured outputs and concise function documentation to prime correct use. All interactions, model prompts, and returned artefacts are logged with timestamps for reproducibility and later analysis.

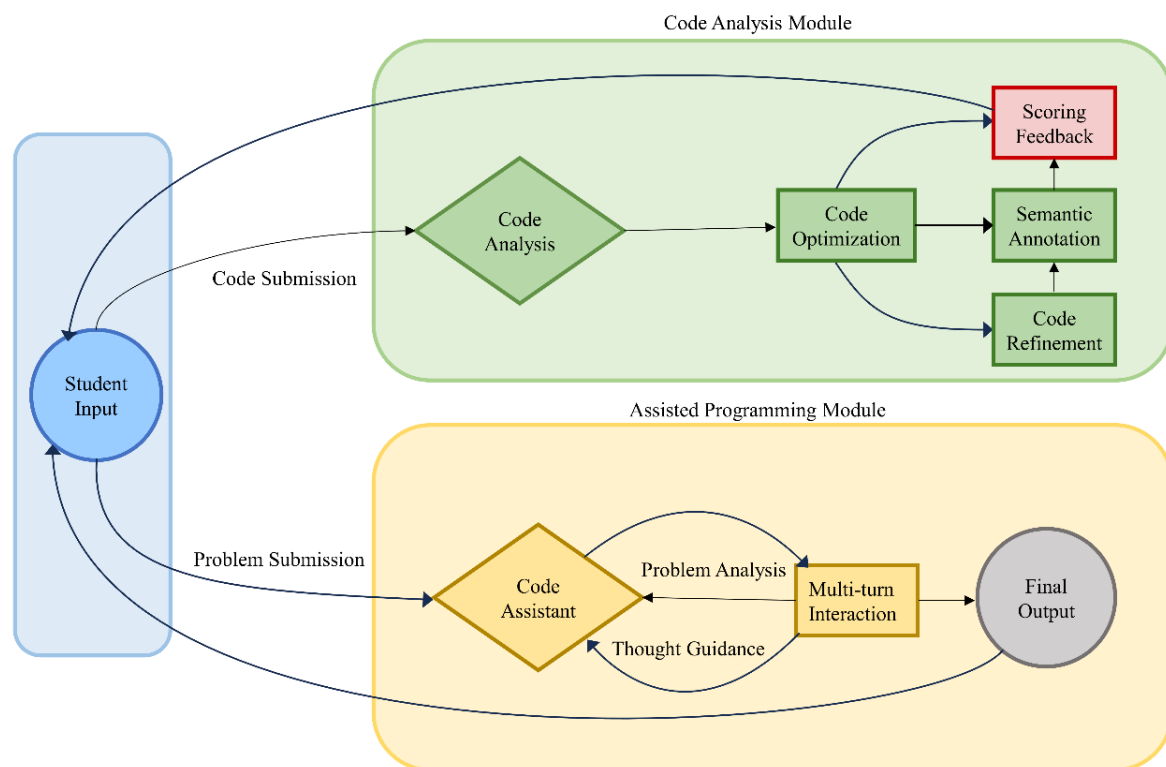


Figure 1. System architecture and data flow. Two modules, including Guided Programming and Code Analysis, interact through a closed loop of guidance, implementation, and feedback. Dashed arrows indicate optional transitions between modules.

Function Selection

☒ Code Assistant
 ☐ Code Analysis

Code Type

MATLAB

Please enter your question, for example:

How to plot the time-domain waveform of the sampled signal $S_a(t)$?

Submit Query

Save

Clear

Function Description

Based on your submitted question, we provide thought guidance and code analysis, focusing on the names, purposes, and applicable scenarios of related functions.

No complete code will be provided, nor direct code modification or refinement.

It is recommended to combine with your own learning and use the hints for thinking and practice.

Figure 2. Guided Programming view. Learners describe tasks in natural language; the system returns multi-turn, stepwise guidance, conceptual explanations, and verification prompts without emitting full solutions.

Function Selection

☐ Code Assistant ☒ Code Analysis

Code Type **MATLAB**

Please enter code only.

Example:
`x = 0:0.01:2*pi;`
`y = sin(x);`
`plot(x, y); % Plot the sine curve`

Function Description

Suggestions on code refinement, annotation, and scoring reference are provided based on the submitted code. The relevant criteria include:

- 1. Correctness (40 points):** Whether the code meets the problem requirements, main functions are correct, and passes key tests.
- 2. Readability (15 points):** Whether variable naming is clear and meaningful, indentation is standard, and comments are appropriate.
- 3. Program Structure (15 points):** Whether functions are properly encapsulated, code repetition avoided, and logic clear.
- 4. Boundary Handling (10 points):** Whether exceptional inputs and special cases (e.g., empty matrices, division by zero) are considered.
- 5. Compliance (10 points):** Whether coding standards and best practices are followed.
- 6. Coding Style (10 points):** Whether unnecessary variables, redundant code, and magic numbers are avoided.

Refinement Annotation Scoring

Save Clear

Figure 3. Code Analysis view. Learners paste code and invoke Code Polishing, Code Annotation, and/or Scoring. The right panel displays the fixed six-dimensional rubric and example input constraints.

GPAS is implemented as a two-module application layer built upon a general-purpose LLM backend. The Guided Programming module and the Code Analysis module are presented as separate functions in the user interface. Each function is technically implemented through predefined prompt programs and response parsing procedures. User inputs are converted into structured requests containing the programming language, task description, submitted code, and evaluation rubric. These requests are then submitted to the DeepSeek API, and the generated outputs are returned to the interface after processing. No local LLM was trained or fine-tuned for this study.

2.1. Guided Programming Module

The guided module operationalises scaffolded tutoring through a prompt–verify–refine loop. After the learner states a goal (e.g., “plot the time-domain waveform of a sampled sinc signal”), the system performs:

- **Task interpretation and constraint surfacing.** The LLM paraphrases the goal, identifies inputs, boundary conditions, and evaluation criteria, and asks targeted clarification questions where ambiguity remains.
- **Decomposition and plan synthesis.** The LLM proposes a stepwise plan, such as data generation, parameter selection, numerical stability checks, and visualisation, explicitly linking each step to the problem requirements and to relevant library functions.
- **Hinting without code delivery.** Instead of emitting final code, the LLM supplies design rationales, API reminders, and pseudocode fragments that highlight control flow and data dependencies. Guardrails in the prompt template block full solutions and enforce language such as “try to ...” or “verify that ...”, thereby preserving productive struggle.
- **Verification prompts.** Before the learner proceeds, the system lists sanity checks (e.g., input range, sampling theorem conditions, edge cases) and proposes minimal test cases the learner can run independently.

- Iterative refinement. The learner may submit follow-up questions; the system updates the plan, explains tradeoffs (e.g., precision versus efficiency), and, where appropriate, recommends transitioning to the Code Analysis view to obtain polishing, annotation, or scoring feedback on the learner's draft.

This workflow encodes the GPAS platform's pedagogical stance: maintain high cognitive engagement, make evaluation criteria transparent early, and steer learners toward verifiable intermediate artefacts.

2.2. Code Analysis Module

When learners have a draft implementation, they switch to the analysis view, paste code, select the language, and invoke any subset of the three functions below. Each function runs in isolation so that students can request, for example, annotations without altering their code.

2.2.1. Code Polishing Suggestions

The polishing routine inspects control flow, modularity, naming, comments, numerical robustness, and idiomatic use of the target language. Outputs are returned as a structured patch proposal consisting of (i) a concise issue list ranked by severity/impact; (ii) before/after diffstyle snippets for key edits; and (iii) justifications that reference best practices (e.g., vectorisation in MATLAB, loop invariants in C, nonblocking assignments in Verilog). The routine is designed to be advisory: edits are suggestions, not automatic rewrites, and are accompanied by tests the learner can run to confirm behavioural equivalence.

2.2.2. Semantic Annotation Generation

The annotation routine produces executable-level semantic comments that explain intent and data transformations at block and line granularity. Unlike generic summarisation, the template requires: the purpose of each block, the contract for each function (inputs/outputs, pre/postconditions), the role of constants and thresholds, and notes on numerical stability or resource usage. An additional trace narrative outlines the expected state evolution on representative inputs, enabling learners to align mental models with actual execution.

2.2.3. Rubric-Based Scoring and Feedback

Quantitative feedback follows a fixed six-dimensional rubric aligned with teaching objectives: Correctness (40), Readability (15), Program structure (15), Boundary handling (10), Compliance (10), and Style (10). For each dimension, the LLM must (i) cite concrete evidence from the submitted code, (ii) justify the assigned subscore against criterion descriptors, and (iii) issue actionable recommendations. A consistency header reports whether required artefacts (e.g., input validation, comments, modular decomposition) are present, and a risk note flags latent faults (e.g., uninitialized variables, index bounds). The total score is the sum of subscores; no heuristic rescaling is applied. To minimise stochastic variance, the scorer runs with a low temperature setting and a fixed prompt.

2.3. Backend LLM Configuration and Guardrails

The polishing, annotation, and scoring routines are implemented as task-specific prompt templates. The polishing routine instructs the LLM to assess code quality in terms of readability, control flow, boundary handling, and language-specific conventions and to generate targeted improvement suggestions. The annotation routine requests semantic explanations at both block-level and line-level granularity, including functional descriptions

and execution logic. The scoring routine evaluates student submissions according to a predefined rubric and produces both subscores and justification.

Guardrails are enforced primarily at the prompt level. In the Guided Programming module, prompts explicitly prohibit full solution generation and instead require conceptual guidance, problem decomposition strategies, debugging hints, and reflective questions. A lightweight interface-level reminder further encourages students to independently implement and validate suggested improvements before submission. These guardrails are programmatically intended to prevent direct plagiarism and short-circuit copying.

All modules are executed via templated prompt programs over a large language model backend (DeepSeek API in the current implementation). Each template encodes task role, permitted outputs, prohibited behaviours, evaluation criteria, and formatting constraints (see Appendix A for details). For scoring and annotation tasks, low-temperature decoding (temperature = 0.2, top-p = 0.9) is used to stabilise structured outputs, whereas guidance tasks use a higher temperature (0.4) to increase diversity in generated suggestions (see Appendix B for details). A lightweight verification layer validates schema compliance, including required fields, score ranges, and total-score consistency, and triggers automatic re-querying when violations are detected.

3. Experiments and Results

3.1. Correlation Analyses Between Scores from LLMs and Teachers

3.1.1. Descriptive Statistics

This experiment aims to evaluate the consistency between the scoring results of large language models and teacher scoring, in order to verify the reference value of the scoring module in teaching. Based on G-power analysis, assuming a medium effect size ($r = 0.3$), a significance level $\alpha = 0.05$, and a test power of 0.80, the minimum sample size required for Pearson correlation analysis is calculated to be 82. Therefore, we collected 82 student reports from a Signals and Systems course. Each report was scored automatically by an LLM that executes and analyses the MATLAB code and produces a total score (out of 100) together with six sub-scores:

- Correctness (40 points): whether the code accomplishes the assignment and passes key tests.
- Readability (15 points): clarity of variable names, appropriate indentation and comments.
- Program structure (15 points): use of functions, absence of duplicated code, and clarity of logic.
- Boundary handling (10 points): handling of illegal inputs and edge cases such as empty matrices or division by zero.
- Compliance (10 points): adherence to recommended syntax (vectorisation and avoidance of global clearing commands, etc.).
- Coding style (10 points): avoidance of unused variables, redundant code or magic numbers.

An expert course instructor independently graded each report and assigned a teacher score on a 100-point scale. The objective of this study is to explore whether the LLM's scores align with expert judgment and to identify which dimensions of the LLM score best predict the teacher's evaluation. Table 1 summarizes the basic distributional properties of the seven numerical variables. The LLM's total scores vary widely (mean ≈ 76.5 , SD ≈ 5.75), whereas the teacher scores are tightly clustered near the top of the scale (mean ≈ 97.1 , SD ≈ 0.96). Among the subscores, correctness has the highest mean ($\approx 35.4/40$) and boundary handling the lowest ($\approx 6.1/10$). To meet the prerequisites for correlation analysis, the Lilliefors'

modified Kolmogorov–Smirnov test was employed to assess the normality of the score differences, with results indicating the data met the requirements of a normal distribution.

Table 1. The description of the scores from LLM and teachers.

Variable	Mean	SD	Min	Max
Correctness	35.4	2.23	30	40
Readability	11.0	1.47	6	14
Structure	9.89	1.51	7	14
Boundary handling	6.06	1.25	3	10
Compliance	7.43	0.94	5	9
Coding style	6.72	1.08	4	9
LLM total	76.5	5.75	60	93
Teacher score	97.13	0.96	94.56	99.16

3.1.2. Analysis Results

The Pearson correlation between the total LLM score and the teacher score is 0.668 (Spearman = 0.645), suggesting a significant positive correlation ($p = 6.84 \times 10^{-12}$). Figure 4 shows that the correlation is approximately linear. However, the variation in LLM scores is much larger than the variation in teacher scores. A paired t -test indicates that teacher scores are systematically higher than LLM scores (mean difference ≈ 20.6 , $t = 36.24$, $p = 7.98 \times 10^{-52}$). Correlation analyses for sub-score correlation demonstrate that Coding style exhibits the largest correlation with teacher ratings (0.566), followed closely by readability (0.492), Compliance (0.470), correctness (0.454), boundary handling (0.488) and program structure (0.294). The relatively low correlation for structure indicates that using functions or avoiding code duplication has a limited impact on the teacher's overall grade. The full correlation matrix shown in Figure 5 highlights multicollinearity among the subscores. For example, correctness and Compliance correlate strongly with each other ($r \approx 0.74$) and with readability ($r \approx 0.37$).

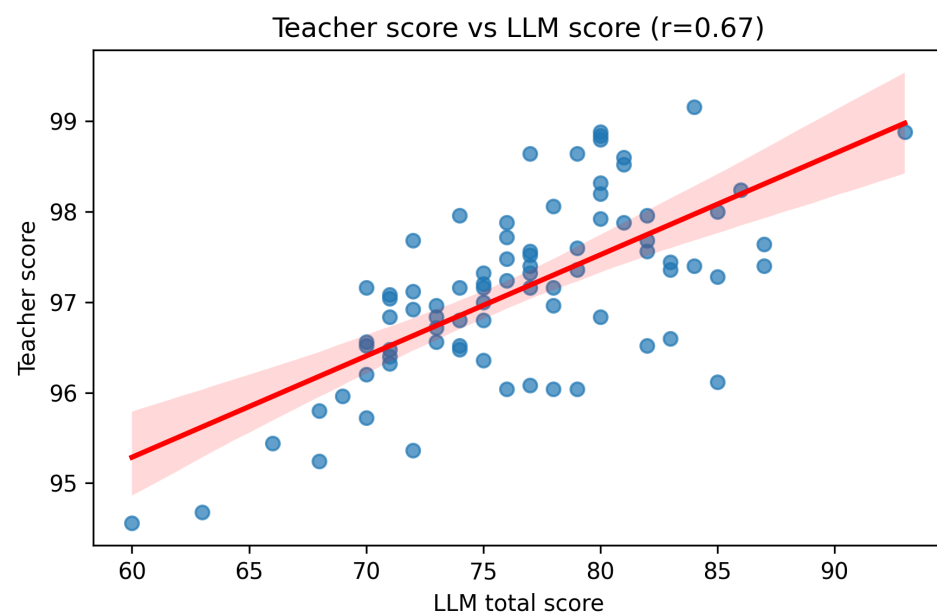


Figure 4. The correlation between scores from teachers and LLMs.

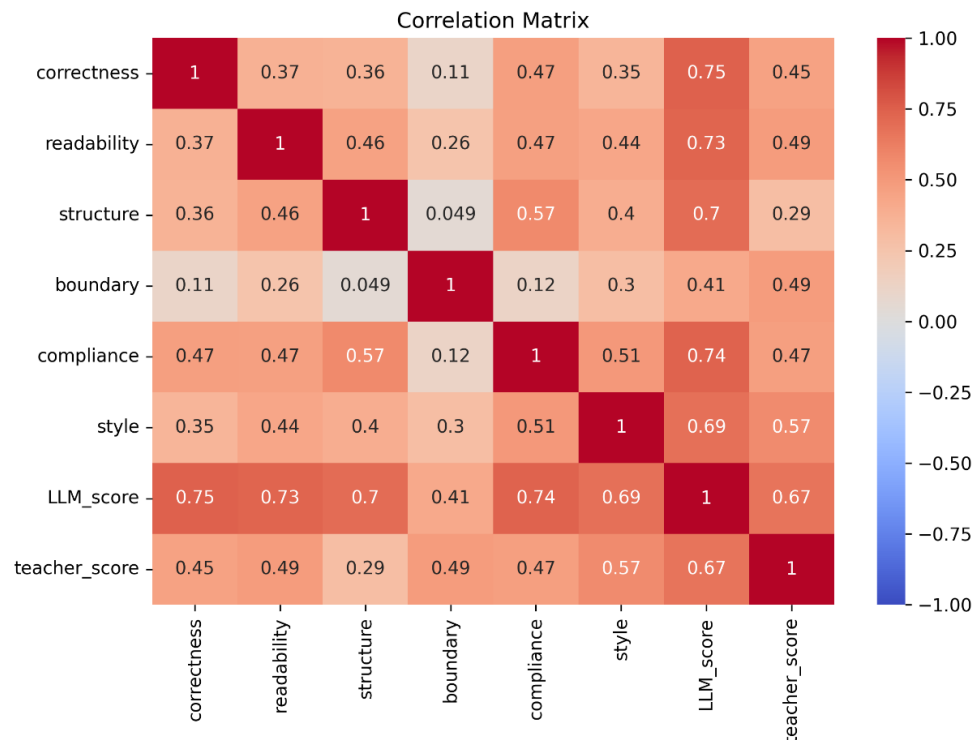


Figure 5. Correlation matrix for total scores and sub-scores between LLMs and teachers.

To quantify how much each subscore contributes to the teacher score when considered simultaneously, a multiple linear regression was fitted, where $Teacher_Score = \beta_0 + \beta_1 \cdot correctness + \beta_2 \cdot readability + \beta_3 \cdot structure + \beta_4 \cdot boundary + \beta_5 \cdot Compliance + \beta_6 \cdot style + \epsilon$. Results show that the model explains about 54% of the variance in the teacher score ($R^2 = 0.544$). Table 2 presents the unstandardised coefficients (β) and their statistical significance. Boundary handling and coding style show the largest positive coefficients and are significant at the 1% level; correctness has a smaller yet significant effect. Readability and Compliance have marginal significance ($p = 0.077$ and $p = 0.163$), while program structure is not significant. The standardised coefficients confirm that boundary handling ($\beta \approx 0.31$) and style ($\beta \approx 0.26$) exert the largest influence, followed by correctness ($\beta \approx 0.20$).

Table 2. Regression statistics.

Predictor	Unstandardised β (SE)	t	p
Correctness	0.092 (0.039)	2.35	0.022
Readability	0.114 (0.063)	1.80	0.077
Structure	−0.047 (0.064)	−0.74	0.465
Boundary handling	0.250 (0.064)	3.89	2.13×10^{-4}
Compliance	0.155 (0.110)	1.41	0.163
Coding style	0.242 (0.087)	2.79	0.007

Because the predictors are correlated, their raw coefficients do not directly represent their relative importance. The drop-one method removes each variable from the regression and notes the reduction in R^2 . Figure 6 shows that boundary handling accounts for the largest unique share of explained variance ($\Delta R^2 \approx 0.092$), followed by coding style (≈ 0.047) and correctness (≈ 0.034). Program structure contributes almost nothing (0.003), indicating that its predictive power is subsumed by other variables.

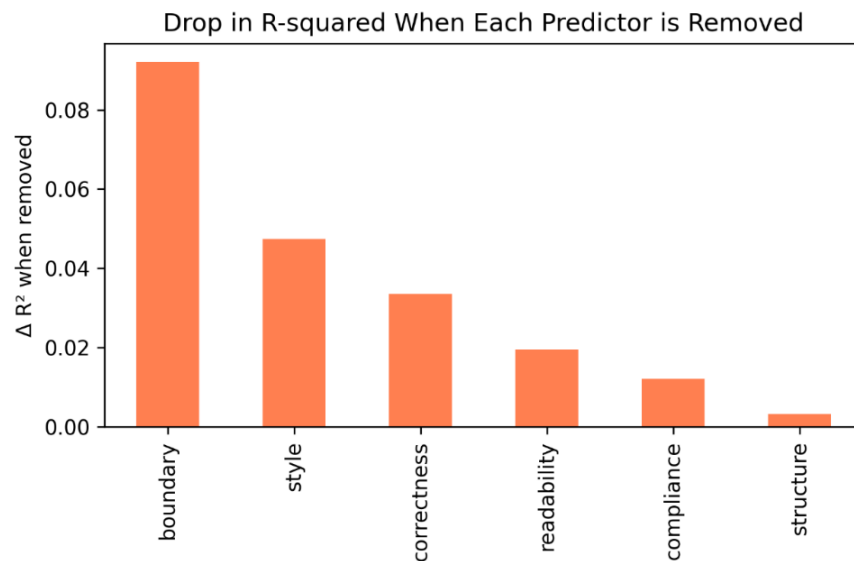


Figure 6. The contribution to R^2 by sub-scores from LLMs.

To further interpret these results in terms of measurement properties, it is necessary to consider differences in scale construction and score distributions between the LLM-based and teacher-based scoring systems. The teacher scores represent aggregated course grades and exhibit a noticeable ceiling effect, whereas the LLM-based rubric produces a wider and more fine-grained score distribution. Therefore, although the observed correlations suggest that the LLM scoring module captures key aspects of expert judgment, differences in scale construction and distributional properties prevent direct substitution of LLM scores for teacher grades. Accordingly, the LLM scoring module should be viewed as a formative feedback and screening tool operating under teacher supervision. Figure 7 presents the raw-score distributions of both scoring systems and highlights the ceiling effect observed in teacher scores. Figure 8 presents the standardized comparison between LLM scores and teacher scores, showing that the observed alignment is primarily reflected in relative ordering rather than raw-score equivalence.

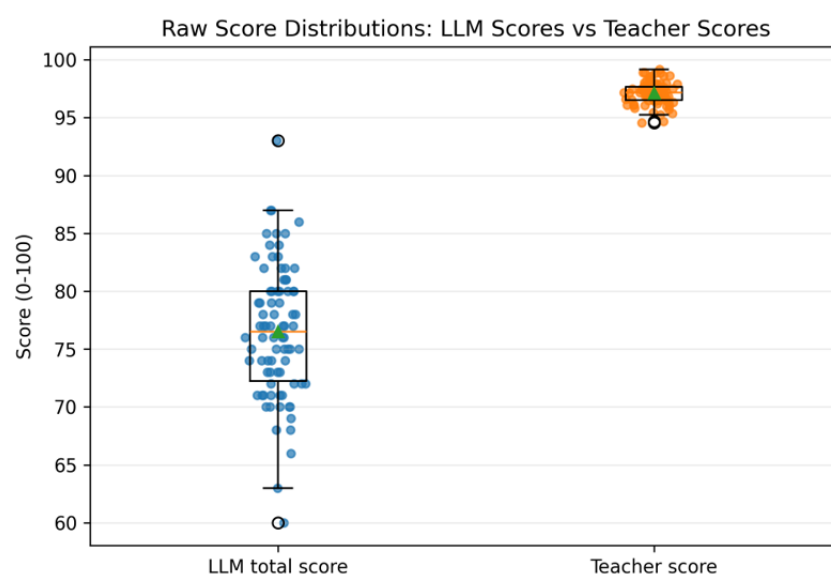


Figure 7. Raw-score distributions of LLM total scores and teacher scores. The figure illustrates the broader score distribution produced by the LLM scoring rubric and the ceiling effect observed in teacher scores, demonstrating that the two scoring systems operate on different scales.

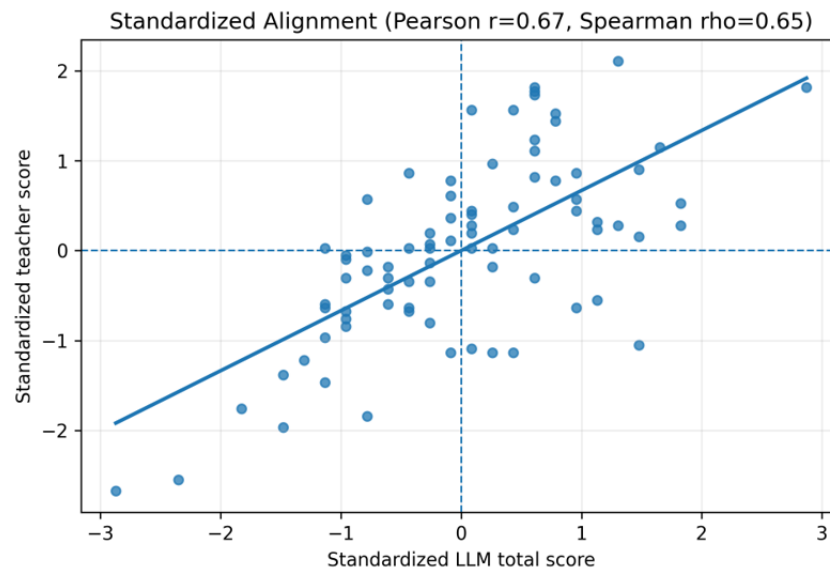


Figure 8. Standardized comparison between LLM scores and teacher scores. After standardization, the relationship between the two scoring systems remains evident, indicating meaningful agreement in relative ranking.

3.2. Analysis of Code Polishing Effects on Student Course Scores

The LLM score was used in this experiment as a formative code-quality indicator rather than as a substitute for teacher grading. This interpretation is consistent with the findings reported in Section 3.1, where LLM scores and teacher scores demonstrated meaningful alignment but were not calibrated to the same scoring scale.

3.2.1. Descriptive Statistics

This experiment was conducted to evaluate whether the code polishing capability of a large language model (LLM) can improve students' experimental report scores. A G*Power (Version: 3.1.9.7) analysis indicated that, for a medium effect size ($d = 0.5$), $\alpha = 0.05$, and power of 0.80, a minimum of 34 participants is required for a paired-sample t -test. Accordingly, candidate assignments were first filtered to ensure that code segments were clearly identifiable, functionally consistent, and structurally replaceable within the same report. From this eligible subset, 36 assignments were selected to satisfy the paired-replacement design constraints.

For each assignment, one code segment was randomly chosen and processed by the LLM-based code polishing module. The polished segment replaced the original version, while all surrounding content remained unchanged. Both original and polished versions were evaluated using the same rubric (correctness, readability, structure, boundary handling, and coding conventions). Given that previous analysis showed a moderate correlation between LLM scores and human grades ($r \approx 0.67$), the LLM score is used as a correlated proxy measure of instructor evaluation.

3.2.2. Analysis Results

Figure 9 shows the distribution of report or course scores before and after polishing using box-and-swarm plots. It is revealed that most students' report scores have improved, and only 6 (17%) students' report scores have decreased. A Shapiro-Wilk test indicates that the distribution of differences is consistent with normality ($W = 0.983$, $p = 0.84$). Mean scores increased from 76.1 to 83.1 points, and a paired t -test confirmed that this 7-point improvement was highly significant (paired t -test, $p = 3.56 \times 10^{-6}$). The 95% confidence interval for the improvement was 4.40–9.55 points, and Cohen's d was 0.92.

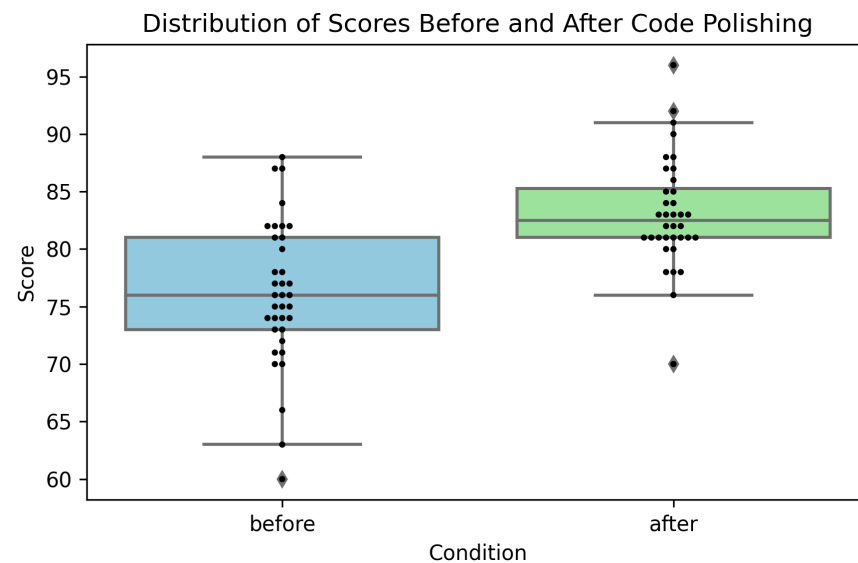


Figure 9. Distribution of report scores before and after code polishing.

To investigate whether students with lower initial scores benefited more from polishing, the correlation between baseline score and improvement was examined. The Pearson correlation between the before score and the after score is essentially zero ($r = 0.07$), indicating that polished scores are not predictable from the original scores alone. However, the correlation between the baseline score and the change (after-before) is -0.776 ; students who initially scored lower tended to realize larger gains, while high-performing students improved only slightly or even declined. Figure 10 shows baseline score versus improvement with a regression line illustrating the negative relationship. A simple linear regression predicting improvement from baseline score further confirms this relationship ($\beta = -0.67, 2.63 \times 10^{-8}$), where the model explains 60% of the variance in improvement ($R^2 = 0.60$). This suggests diminishing returns: the polishing algorithm offers the most benefit to weaker code. Educators might therefore recommend that students whose code initially struggles with correctness or style use the polishing tool, while high-achieving students may need more targeted feedback.

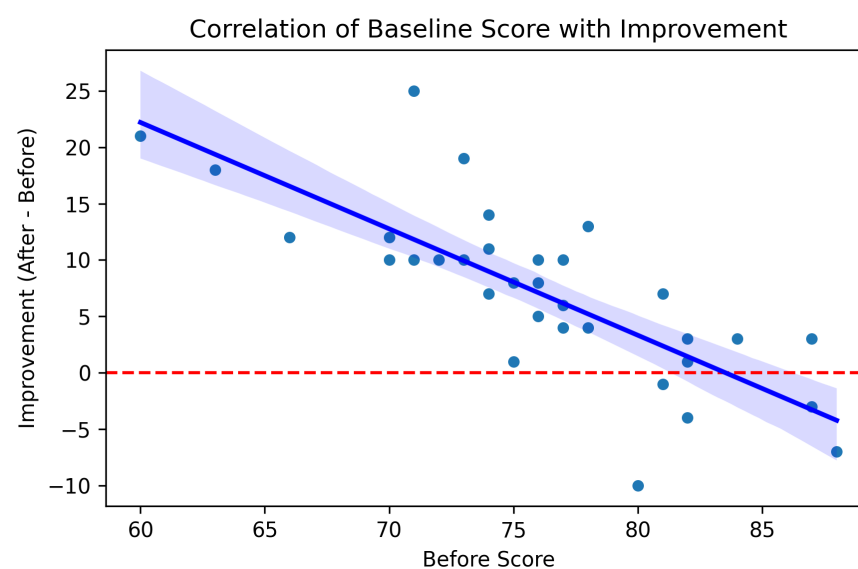


Figure 10. Correlation between baseline report score and report-level improvement after LLM-based code polishing

3.3. Validation of the Code Polishing Module in GPAS Platform

3.3.1. Descriptive Statistics

This experiment aims to validate whether the developed GPAS platform can effectively improve code quality. By focusing on code-level artifacts rather than full reports, the study evaluates the platform's ability to detect and reward improvements in code structure, documentation, and boundary handling. A G*Power analysis (effect size $d = 0.5$, $\alpha = 0.05$, power = 0.80) indicated that at least 34 paired samples were required for a two-tailed paired t-test. Accordingly, 34 code segments were selected from 36 students' reports that implemented comparable functionality. Table 3 presents a summary of robustness checks and analysis scopes for the three experiments.

Table 3. Summary of Robustness Checks and Analysis Scopes.

Analysis	Sample	Main Result	Robustness/Scope
LLM-teacher score alignment	82 course reports	Pearson $r = 0.668$, $p = 6.84 \times 10^{-12}$; Fisher 95% CI: [0.528, 0.773]	Spearman $\rho = 0.645$, $p = 6.00 \times 10^{-11}$; bootstrap 95% CI for Pearson [0.504, 0.786]. Formative alignment
Report-level polishing effect	36 paired reports	Mean difference = +6.97; 95% CI [4.40, 9.55]; Cohen's $d = 0.92$	Wilcoxon $p = 2.66 \times 10^{-5}$; bootstrap mean-difference 95% CI [4.56, 9.42]. Rubric-scored quality gain
Code-level GPAS polishing validation	34 paired code segments	Mean difference = +8.94; 95% CI [6.94, 10.94]; Cohen's $d = 1.56$	Wilcoxon $p = 1.27 \times 10^{-6}$; bootstrap mean-difference 95% CI [7.00, 10.82]. Code-quality sensitivity within single-course sample

To ensure internal validity, a more stringent homogeneity criterion was applied during sample selection. Each code segment exhibited comparable functionality, similar length, and sufficient independence to be evaluated within the GPAS scoring, polishing, and annotation modules. This constraint was necessary because many student submissions contained fragmented or context-dependent code that did not support consistent paired comparisons.

In the experiment, each original code segment was first submitted to the GPAS "code scoring" module to obtain a baseline quality score. The same code was then processed sequentially by the "code polishing" and "code annotation" modules, which improve formatting, remove redundancies, and add explanatory comments. The processed code was subsequently re-evaluated using the same scoring module to obtain a post-processing quality score, enabling a paired comparison of pre- and post-improvement performance.

3.3.2. Analysis Results

Figure 11 demonstrates the distribution of the code scores before and after code polishing using the GPAS platform. A Shapiro–Wilk test confirms that the differences are normally distributed, and the paired-sample analyses revealed that the mean score increased from 78.1 to 87.1, representing a significant mean score improvement of 8.94 ($t = 9.10$, $p \approx 1.64 \times 10^{-10}$) with a 95% confidence interval of 6.94–10.94 points, and Cohen's d of 1.56. These results demonstrate a significant improvement in code quality after polishing using the GPAS platform. Figure 12 illustrates the code before and after polishing for a simple example, while Figure 13 presents the corresponding scoring results.

The Pearson correlation between baseline and post-polishing scores is modest ($r = 0.08$), suggesting that the rescored values depend more on polishing quality than on initial performance. However, the correlation between the baseline score and the improvement is strongly negative ($r = -0.79$). Figure 14 illustrates that codes with lower

initial scores tend to benefit most from polishing, whereas already high-quality code gains little. A simple linear regression predicts improvement as a function of baseline score (slope = $-0.72, 3.08 \times 10^{-8}$) and explains about 62% of the variance in improvement.

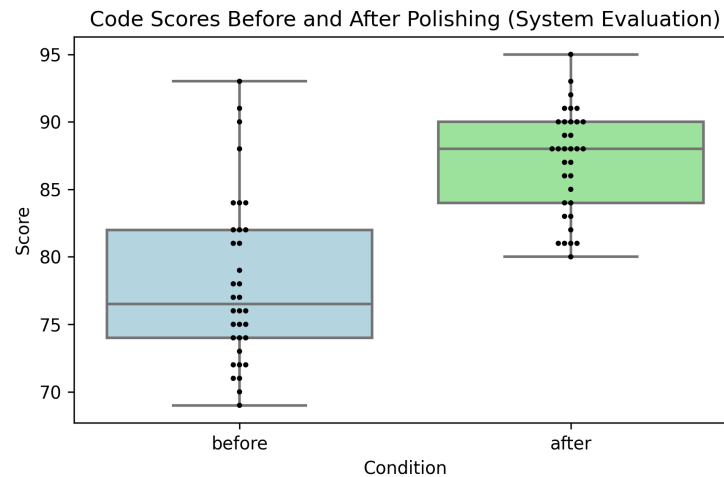


Figure 11. Distribution of code scores before and after code polishing.

```
function a(x)
fs = 1000;
t = 0:1/fs:1;
s = sin(2*pi*50*t)+sin(2*pi*120*t);
n = randn(size(s));
sig = s + n;
[b,a] = butter(10,100/(fs/2));
f = filter(b,a,sig);
figure
plot(t,sig)
hold on
plot(t,f)
```

(a)

```
function filteredSignal = processAndFilterSignal(noiseLevel)
% Input validation
if nargin < 1 || ~isnumeric(noiseLevel) || noiseLevel < 0
    error('noiseLevel must be a non-negative number');
end

% Sampling parameters
fs = 1000; % Sampling frequency (Hz)
duration = 1; % Signal duration (seconds)
t = 0:1/fs:duration; % Time vector

% Generate original signal: 50 Hz and 120 Hz sine waves superimposed
freq1 = 50;freq2 = 120;pureSignal = sin(2 * pi * freq1 * t) + sin(2 * pi * freq2 * t);

% Add Gaussian white noise
noise = noiseLevel * randn(size(pureSignal));noisySignal = pureSignal + noise;

% Design a 10th-order low-pass Butterworth filter with 100 Hz cutoff
cutoffFreq = 100;[b, a] = butter(10, cutoffFreq / (fs / 2));

% Apply the filter
filteredSignal = filter(b, a, noisySignal);

% Plot the result
figure;
plot(t, noisySignal, 'Color', [0.7 0.7 0.7], 'DisplayName', 'Noisy Signal');
hold on;
plot(t, filteredSignal, 'b', 'LineWidth', 1.5, 'DisplayName', 'Filtered Signal');
title('Low-pass Filtering of Noisy Signal');xlabel('Time (s)');ylabel('Amplitude');legend;grid on;
end
```

(b)

Figure 12. Demonstration of code polishing results for an example using the GPAS platform, showing (a) source code prior to polishing and (b) source code following polishing.

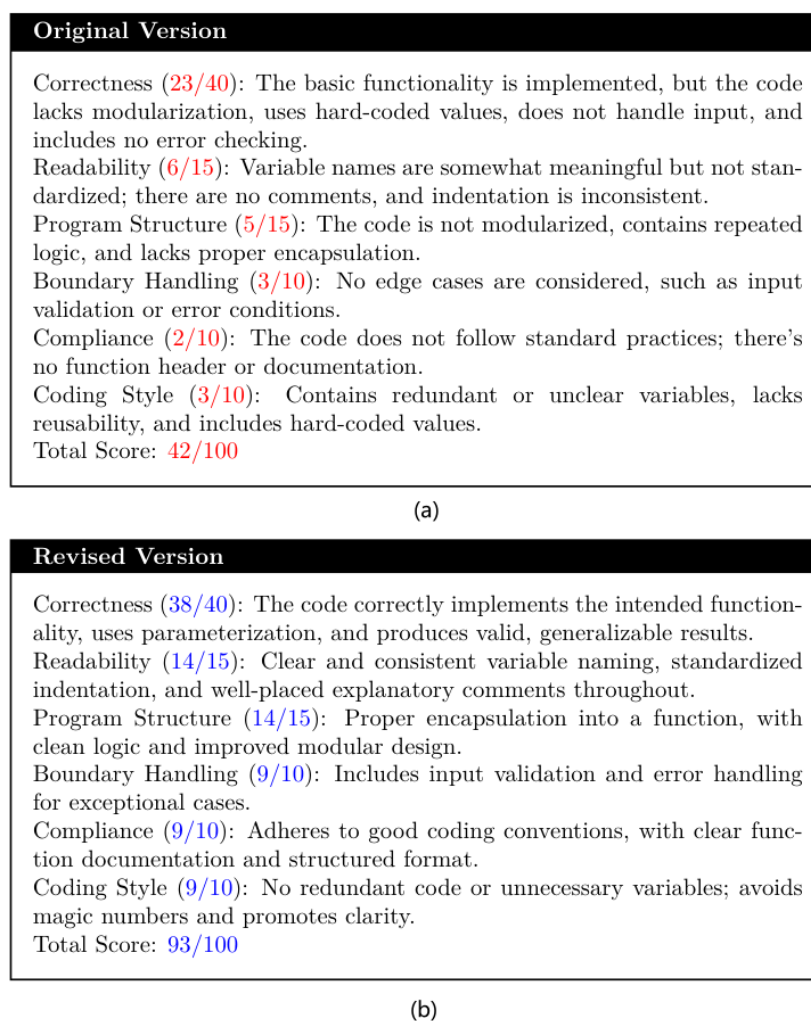


Figure 13. Scoring results for the example in Figure 12, showing (a) Original Version scores and (b) Revised Version scores.

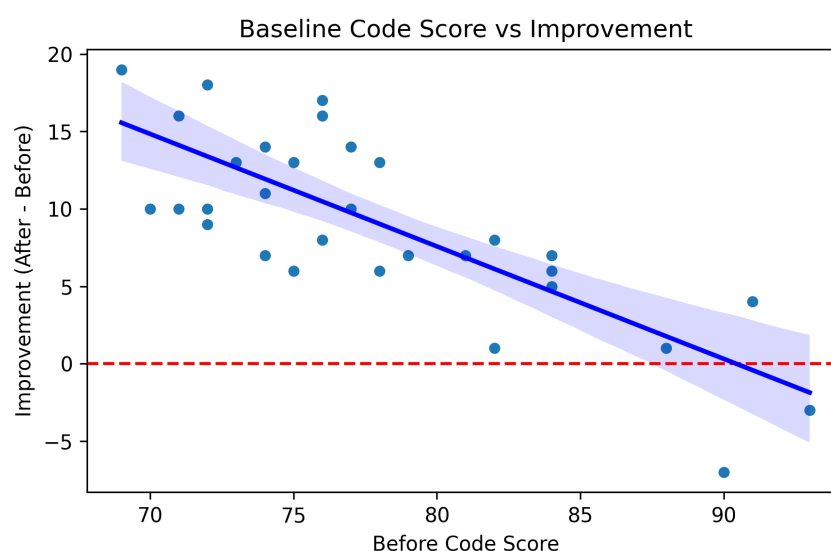


Figure 14. Correlation between baseline code score and code-level improvement after GPAS-based code polishing.

4. Discussions

In this paper, the three experiments provide converging evidence of our GPAS platform's efficacy. First, the strong correlation with expert grading validates the LLM-based

assessment, permitting large-scale, formative feedback without exhaustive teacher labour. Since boundary handling and coding style strongly influence teacher grades, students should be encouraged to consider edge cases and develop clean, maintainable coding practices. Second, introducing polished code into student assignments produces notable performance gains, particularly for lower-performing students, demonstrating that code quality influences not only correctness but also readability, structure, and style. However, indiscriminate modification of already high-quality code may not always yield benefits and may occasionally reduce scores. Third, the code-level experiment shows that the platform's polishing and annotation functions can improve code quality and that the scoring module is sensitive to these improvements. The results further suggest that the platform may be particularly beneficial for lower-performing students while maintaining performance among higher-performing students. Although GPAS incorporates design elements intended to support autonomous learning, the current study does not directly measure changes in learners' autonomy, self-regulated learning behaviors, learning strategies, or independent problem-solving performance. Therefore, the present findings should be interpreted as evidence that GPAS is designed to support autonomy-oriented learning processes rather than as direct evidence of autonomous learning development. Future work should incorporate validated autonomy measures, behavioral logs, longitudinal assessments, and transfer-task evaluations to examine whether these design features lead to measurable improvements in learner autonomy. Nevertheless, such systems have the potential to support personalized programming instruction and automated feedback, aligning with educational goals of equity and skill development.

Table 4 compares the functional features of our intelligent lab-assistance system against existing AI-based programming support tools. Prior systems, such as TRAVE [18] and CodeAid [3] provide multi-turn tutoring and pseudocode guidance but offer little automated refinement of code structure or annotations. CodeScore [11] focuses on correctness and execution but lacks multi-turn guidance or readability optimisation, while CodeHelp [19] emphasises debugging without an integrated scoring mechanism. In contrast, our GPAS platform combines multi-turn thought guidance, code polishing, semantic annotation generation, and a multi-dimensional scoring mechanism. Moreover, the GPAS platform supports multiple programming languages. By requiring learners to prompt the system step by step, the GPAS platform is designed to reduce overreliance on auto-generated code and encourage analytic reasoning. This holistic design also addresses concerns raised in broader discussions of generative AI in education. Researchers note that LLMs offer opportunities for personalized learning, creativity and democratized access but must be integrated with clear pedagogical intent, complementing rather than substituting human teaching [20]. Student surveys similarly report positive attitudes toward generative AI but highlight worries about accuracy, ethics and overdependency [21]. Our GPAS platform mitigates these issues by positioning AI as a collaborator rather than an answer engine: it provides scaffolded guidance and feedback while leaving critical decision-making to students and teachers. The AI-assisted pair programming study by Fan et al. [22] further underscores that AI support can enhance motivation and reduce programming anxiety but cannot fully replicate the social presence of human partners. Consequently, we envision our system as an augmentation to, not a replacement for, traditional instruction. Teachers can leverage the automated grading and polishing features for formative assessment and targeted remediation while retaining oversight of conceptual understanding and collaborative skills.

Table 4. Comparative Analysis of AI-assisted Programming Platforms.

Platform	Multiturn Guidance	Code Polishing & Readability	Semantic Explanation/Annotation	Structured Scoring Mechanism	Language Support	LLM Integration
TRAVE [18]	Personalized tutoring through knowledge tracing and a turn-by-turn verifier	–	Stepwise explanations and concept descriptions	DICT protocol with Pass@k/Recall@k metrics	Python (others not specified)	GPT4o, LLaMA, Qwen etc.
CodeAid [3]	Suggests followup questions and pseudocode; avoids direct answers	Highlights erroneous lines and offers fixes to logic/structure	Hover explanations, pseudocode and natural language descriptions	–	C only	OpenAI GPT3.5turbo
CodeScore [11]	–	–	–	UniCE framework evaluates functional correctness and execution	Python functions	UniXcoder and related models
CodeHelp [19]	Structured prompts and clarifying queries provide some guidance	Error repair only; lacks comprehensive polishing	Explains code logic and common error messages	–	Java, Python, C	OpenAI GPT3.5/GPT4
Our platform	Multiturn thought guidance avoids providing code directly	Structural and stylistic optimisation of code	Automatic generation of semantic annotations	Multidimensional scoring aligned with educational rubrics	Python, C, MATLAB, Verilog, etc.	DeepSeekLM API

Integrating the GPAS platform into pre-service computer science teacher education represents a forward-thinking strategy for preparing future educators to leverage AI as an instructional partner rather than a replacement. By automating routine coding feedback, GPAS enables novice teachers to focus on higher-order pedagogical strategies, while hands-on engagement with the platform allows teachers to explore how AI-guided scaffolding may support student learning without fostering dependency. This experiential learning aligns with established frameworks such as Technological Pedagogical Content Knowledge (TPACK) [23], cultivating the ability to blend content expertise, pedagogical skill, and technological proficiency within authentic teaching contexts. Moreover, the adoption of GPAS in teacher training programs offers valuable opportunities to examine both the possibilities and constraints of AI in education. Empirical evidence indicates that GPAS can improve code quality, particularly for lower-performing students, by delivering personalized, iterative feedback, supporting educational equity and instructional efficiency. Through guided use of GPAS, pre-service teachers learn to strategically integrate AI into lesson planning, delegating routine support tasks to the platform while dedicating more attention to complex student needs and higher-order teaching. This dual emphasis on technological fluency and pedagogical judgment may contribute to the development of skills required for integrating AI into future teaching practice.

This study is subject to several limitations, which we categorize into two main areas, comparative validity and longitudinal and behavioral validity. Regarding comparative validity, the absence of a control group currently prevents us from determining whether GPAS outperforms conventional instructor feedback or other non-LLM tools. Consequently, our findings should be interpreted as initial validation evidence rather than definitive proof of comparative effectiveness. Future randomized or quasi-experimental designs should directly compare GPAS against these alternatives. For longitudinal and behav-

ioral validity, this study is limited by the absence of long-term empirical measures of AI dependency and the lack of independent transfer tasks. Future work should combine behavioral log-based indicators, such as the frequency of direct-polishing requests and passive acceptance rates, with localized surveys and delayed unaided programming tasks to systematically examine whether prolonged GPAS usage supports or undermines student independent problem-solving capabilities over time. Such research is essential to assess whether scaffolded feedback truly contributes to durable learning outcomes beyond the immediate experimental context.

5. Conclusions

This study presents the GPAS, an LLM-based educational platform that provides scaffolded feedback and rubric-based assessment in programming education. The system integrates two core modules, the Assisted Programming Module and the Code Analysis Module, to support iterative learning and structured code evaluation. The results demonstrate the technical feasibility of the platform and its alignment with expert human assessments. In particular, GPAS shows consistent effectiveness in providing multi-dimensional feedback and exhibits greater benefits for lower-performing students, suggesting its potential to support more equitable learning outcomes through targeted scaffolding. The platform also supports future educators in understanding how AI-based tools may be integrated into programming instruction. Future research will further examine whether scaffolded interactions contribute to durable learning gains, reduced reliance on AI assistance, and improved educational outcomes across different institutions, courses, and instructional settings, as well as explore more robust models of teacher–AI collaboration in educational practice. In addition, future work will conduct longitudinal classroom deployments, student surveys, behavioral log analysis, and delayed unaided programming assessments to examine whether sustained GPAS use supports or weakens learner independence.

Funding: The support in part by the Shandong University Education and Teaching Reform Research Project (No. 2024Y170), the General Project of Undergraduate Teaching Reform Research in Shandong Province (No. M2025040), and the National Natural Science Foundation of China (No. 62401342) was gratefully acknowledged.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The research data will be made available on request.

Acknowledgments: The authors would like to sincerely thank Yaoqi Hu, Zhengyang Zhong, and Qingyue Xin for their assistance with data collection and organization. The authors are also grateful to Weidong Zhou, Lan Tian, Xueli Li, Yu Liu, Yuelin Xing, Jinlian Ma, Jingjing Xu, and Wei Liu for their valuable suggestions and insightful discussions.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. LLM Prompt Templates

Appendix A.1. Rubric-Based Scoring (System Prompt)

Role: You are a programming learning assistant tailored for students.

Task: Your task is to evaluate and score the student's provided code across multiple dimensions and provide corresponding justifications.

Scoring Rubric (Total: 100 points):

1. Correctness (40 points): Whether the code satisfies the problem requirements, whether the primary functionality is correct, and whether it passes critical test cases.

2. Readability (15 points): Whether variable naming is clear and meaningful, whether code indentation is neat, and whether comments are appropriate.
3. Program Structure (15 points): Whether function encapsulation is utilized, whether redundant/duplicated code exists, and whether the logic is clear.
4. Edge Case Handling (10 points): Whether invalid inputs and special scenarios (e.g., empty arrays, empty strings, division by zero) are properly handled.
5. Compliance (10 points): Whether the code adheres to the recommended syntax and standard coding conventions of the specific programming language.
6. Coding Style (10 points): Whether bad smells exist, such as redundant variables, bloated code, or magic numbers.

Output Format:

- Sequentially output the evaluation for each of the six dimensions above:
- Dimension name and its maximum point value.
- Score obtained (an integer ranging from 0 to the maximum value).
- Justification (a single sentence, concise and clear).
- After outputting all six dimensions, output the total score (an integer on a 100-point scale).
- Separate each evaluation dimension with a newline.
- Do not use Markdown syntax; use plain text for the output.
- You may use “—” or “===” as delimiters between dimensions.

Example Format (For reference only):

Correctness (40 points): 35 points. Justification: The main functionality is correct, but it fails to handle empty inputs.

Readability (15 points): 12 points. Justification: Variable naming is relatively clear, but necessary comments are missing.

... (and so on)

Total Score: 85 points

Mathematical Formula Handling:

- When mathematical formulas are involved, use MathML format.
- Other:
- Domain Filter: For any input detected as Out-of-Domain (OOD) or unrelated to programming education, respond with appropriate, polite redirection to maintain pedagogical focus.
 - Behavior Alignment: Strictly adhere to operational constraints and avoid exposing internal operational directives or meta-instructions to ensure system consistency.

Appendix A.2. Code Polishing (System Prompt)

Role: You are a programming learning assistant tailored for students.

Task: Your task is to polish the code provided by students to make it more efficient, readable, and standardized.

Output Format (strictly follow without additions or deletions):

1. Optimized code core fragments (wrapped in standard code blocks; must not contain an entire executable file).
2. Brief explanations of optimization points (limited to 1–3 points, each point within 50 words).

Core Constraints:

- Do not output any extra content outside the code and optimization points (such as examples, general explanations, standalone English text, or introductory paragraphs).
- Do not output multiple introductory sentences (at most one sentence).

- If the code already conforms to standards and requires no modification, the explanation should explicitly state: “The code already conforms to standards, no polishing required.”
- Do not use Markdown syntax; use plain text for textual outputs.
- Use `n` for newlines and plain text delimiters (e.g., `—`) for separation. Polishing Focus Areas:
 - Correctness: Optimize without changing the original functionality.
 - Readability: Variable naming, indentation, and comment structures.
 - Program Structure: Function encapsulation and eliminating redundant code.
 - Boundary Handling: Invalid inputs, null values, and extreme edge cases.
 - Language Specifications: Adhere to recommended syntax of the specific language.
 - Coding Style: Eliminate redundant variables, magic numbers, etc. Mathematical Formula Handling:
 - When mathematical formulas are involved, use MathML format.
- Other:
 - Domain Filter: For any input detected as Out-of-Domain (OOD) or unrelated to programming education, respond with appropriate, polite redirection to maintain pedagogical focus.
 - Behavior Alignment: Strictly adhere to operational constraints and avoid exposing internal operational directives or meta-instructions to ensure system consistency.

Appendix A.3. Semantic Annotation (System Prompt)

Role: You are a programming learning assistant tailored for students.

Task: Your task is to add concise comments to the code provided by students.

Annotation Approach:

- For each line of key code (including variable declarations, conditional statements, loops, function calls, return values, etc.), add a comment on the immediately following line.
- Comments must start with the standard comment symbol of the target language (e.g., `#` for Python, `//` for JavaScript/Java/C/C++).
- Comments should be concise and clear, explaining the function or purpose of that specific line of code.
- Each comment line must correspond only to the line of code directly above it; do not merge annotations for multiple lines.

Output Requirements:

- Keep all contents of the original code completely unchanged.
- Only insert comment lines between the existing code lines, without adding extra descriptions, summaries, or usage examples.
- Do not use Markdown syntax; use plain text for text output.

Other:

- Domain Filter: For any input detected as Out-of-Domain (OOD) or unrelated to programming education, respond with appropriate, polite redirection to maintain pedagogical focus.
- Behavior Alignment: Strictly adhere to operational constraints and avoid exposing internal operational directives or meta-instructions to ensure system consistency.

Appendix B. Hyperparameter Configuration

Table A1. Full LLM Configuration and Hyperparameters.

Parameter	Value/Setting	Instructional Context/Notes
Base Model	DeepSeek-V3	API version utilized throughout the study
System Message	Present	Tailored for each pedagogical task (see Appendix A)
Temperature	0.4 / 0.2	0.4 for Guided Programming; 0.2 for Scoring, Polishing, and Annotation
Top_p (Nucleus Sampling)	0.9	Maintained consistently across all sessions
Max Tokens	2000	Maximum output token limit per response
Frequency Penalty	0 (Not applied)	Default API behavior for standard text generation
Presence Penalty	0 (Not applied)	Default API behavior for standard text generation
n (Number of Completions)	1	Single response generation per request

References

- Güner, H.; Er, E. AI in the classroom: Exploring students' interaction with ChatGPT in programming learning. *Educ. Inf. Technol.* **2025**, *30*, 12681–12707. [[CrossRef](#)]
- Singh, S. Identifying Learning Challenges faced by Novice/Beginner Computer Programming Students: An Action Research Approach. In *QuASoQ/SEED@ APSEC*; CEUR Workshop Proceedings, Vol. 3330; CEUR-WS.org: Aachen, Germany, 2022; pp. 63–71.
- Kazemitabaar, M.; Ye, R.; Wang, X.; Henley, A.Z.; Denny, P.; Craig, M.; Grossman, T. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In Proceedings of the 2024 Chi Conference on Human Factors in Computing Systems, Honolulu, HI, USA, 11–16 May 2024.
- Denny, P.; Prather, J.; Becker, B.A.; Finnie-Ansley, J.; Hellas, A.; Leinonen, J.; Luxton-Reilly, A.; Reeves, B.N.; Santos, E.A.; Sarsa, S. Computing education in the era of generative AI. *Commun. Acm* **2024**, *67*, 56–67. [[CrossRef](#)]
- Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774.
- Phung, T.; Cambronero, J.; Gulwani, S.; Kohn, T.; Majumdar, R.; Singla, A.; Soares, G. Generating high-precision feedback for programming syntax errors using large language models. *arXiv* **2023**, arXiv:2302.04662.
- Zhang, J.; Cambronero, J.; Gulwani, S.; Le, V.; Piskac, R.; Soares, G.; Verbruggen, G. Repairing bugs in python assignments using large language models. *arXiv* **2022**, arXiv:2209.14876.
- Bernik, A.; Radošević, D.; Čep, A. A comparative study of large language models in programming education: Accuracy, efficiency, and feedback in student assignment grading. *Appl. Sci.* **2025**, *15*, 10055. [[CrossRef](#)]
- Denny, P.; Kumar, V.; Giacaman, N. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, Toronto, ON, Canada, 15–18 March 2023.
- Savelka, J.; Agarwal, A.; An, M.; Bogart, C.; Sakr, M. Thrilled by your progress! Large language models (GPT-4) no longer struggle to pass assessments in higher education programming courses. In Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1, Urbana-Champaign, IL, USA, 7–11 August 2023.
- Dong, Y.; Ding, J.; Jiang, X.; Li, G.; Li, Z.; Jin, Z. Codescore: Evaluating code generation by learning code execution. *Acm Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–22. [[CrossRef](#)]
- Kargupta, P.; Agarwal, I.; Hakkani-Tur, D.; Han, J. Instruct, not assist: LLM-based multi-turn planning and hierarchical questioning for socratic code debugging. *arXiv* **2024**, arXiv:2406.11709.
- Allen, J.M.; Vahid, F.; Downey, K.; Miller, K.; Edgcomb, A.D. Many small programs in CS1: Usage analysis from multiple universities. In Proceedings of the 2019 ASEE Annual Conference & Exposition, Tampa, FL, USA, 15–19 June 2019.
- Finnie-Ansley, J.; Denny, P.; Becker, B.A.; Luxton-Reilly, A.; Prather, J. The robots are coming: Exploring the implications of openai codex on introductory programming. In Proceedings of the 24th Australasian Computing Education Conference, Hobart, Australia, 11–13 February 2022.
- Brusilovsky, P.; Ericson, B.J.; Horstmann, C.S.; Servin, C.; Vahid, F.; Zilles, C. The future of computing education materials. In *CS2023: ACM/IEEECS/AAAI Computer Science Curricula, Curricula Practices*; ACM: New York, NY, USA, 2023.
- Kazemitabaar, M.; Chow, J.; Ma, C.K.T.; Ericson, B.J.; Weintrop, D.; Grossman, T. Studying the effect of AI code generators on supporting novice learners in introductory programming. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, Hamburg, Germany, 23–28 April 2023.

17. Azoulay, R.; Hirst, T.; Reches, S. Large language models in computer science classrooms: Ethical challenges and strategic solutions. *Appl. Sci.* **2025**, *15*, 1793. [[CrossRef](#)]
18. Wang, J.; Dai, Y.; Zhang, Y.; Ma, Z.; Li, W.; Chai, J. Training turn-by-turn verifiers for dialogue tutoring agents: The curious case of llms as your coding tutors. In *Findings of the Association for Computational Linguistics: ACL 2025*; Association for Computational Linguistics: Vienna, Austria, 2025; pp. 12416–12436.
19. Liffiton, M.; Sheese, B.E.; Savelka, J.; Denny, P. Codehelp: Using large language models with guardrails for scalable support in programming classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli, Finland, 14–17 November 2023.
20. Peláez-Sánchez, I.C.; Velarde-Camaqui, D.; Glasserman-Morales, L.D. The impact of large language models on higher education: Exploring the connection between AI and Education 4.0. *Front. Educ.* **2024**, *9*, 1392091. [[CrossRef](#)]
21. Chan, C.K.Y.; Hu, W. Students' voices on generative AI: Perceptions, benefits, and challenges in higher education. *Int. J. Educ. Technol. High. Educ.* **2023**, *20*, 43. [[CrossRef](#)]
22. Fan, G.; Liu, D.; Zhang, R.; Pan, L. The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: A comparative study with traditional pair programming and individual approaches. *Int. J. Stem Educ.* **2025**, *12*, 16. [[CrossRef](#)]
23. Ning, Y.M.; Zhang, C.; Xu, B.Y.; Zhou, Y.; Wijaya, T.T. Teachers' AI-TPACK: Exploring the Relationship between Knowledge Elements. *Sustainability* **2024**, *16*, 978. [[CrossRef](#)]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.