Actes de CORIA-TALN-RJCRI-RECITAL 2025
Marseille, France, 30 juin au 4 juillet 2025

Pedagogical Code Evaluation with Large Language Models A Large Scale Comparative Study against Unit Testing

Anton Conrad[1] Laila Elkoussy[2] Julien Perez[2]
(1) EPITECH, 94270, Le Kremlin-Bicêtre, France
(2) Laboratoire de Recherche d'EPITA, 94270, Le Kremlin-Bicêtre, France
anton.conrad@epitech.eu, laila.elkoussy@epita.fr, julien.perez@epita.fr

Résumé

L'évaluation automatisée en éducation par projet pour l'apprentissage de la programmation s'appuie traditionnellement sur les tests unitaires pour juger les soumissions de code des étudiants, mettant l'accent sur la correction fonctionnelle. Cependant, ces tests négligent souvent des aspects qualitatifs du code, comme la lisibilité ou la modularité. Cette étude examine le potentiel des grands modèles de langage (LLM) pour évaluer les soumissions de programmation, en comparant leurs résultats à ceux des tests unitaires. À partir d'un grand ensemble de données de rendus d'étudiants à une collection de projets de développement logiciel, nous appliquons des analyses statistiques, modélisations prédictives, ainsi que plusieurs comparaisons pour évaluer l'efficacité des LLMs. Nos résultats mettent en évidence une corrélation significative entre les évaluations des LLMs, pour des prompts donnés, et les tests unitaires. Les modèles prédictifs montrent que les scores des LLMs peuvent être approximés à partir des résultats des tests unitaires, et les classements d'étudiants issus des deux approches sont fortement corrélés. Ces constats restent robustes même en présence de bruit injecté dans les rendus étudiants. Ces résultats suggèrent que les LLM, en capturant des dimensions supplémentaires de la performance, peuvent enrichir les cadres d'évaluation éducative, offrant une approche totale plus nuancée et complète.

Abstract

Pedagogical Code Evaluation with Large Language Models, A Large Scale Comparative Study against Unit Testing

Automated evaluation in project-based education for learning programming relies traditionally on unit tests to judge student code submissions, emphasizing functional correctness. However, these tests often neglect qualitative aspects of code, such as readability or modularity. This study examines the potential of large language models (LLMs) to evaluate programming submissions, by comparing their results to those of unit tests. Based on a large dataset of student submissions to a collection of software development projects, we apply statistical analyses, predictive modeling, as well as several comparisons to assess the effectiveness of LLMs. Our results highlight a significant correlation between LLM evaluations, for given prompts, and unit tests. Predictive models show that LLM scores can be approximated from unit test results, and student rankings from both approaches are strongly correlated. These findings remain robust even in the presence of noise injected into student submissions. These results suggest that LLMs, by capturing additional dimensions of performance, can enrich educational assessment frameworks, offering a more nuanced and comprehensive overall approach.

Mots-clés : évaluation automatisée, tests unitaires, grands modèles de langage, qualité du code, évaluation en éducation, analyse statistique

Actes de l'atelier Évaluation des modèles génératifs (LLM) et challenge 2025 (EvalLLM), pages 188–201.

Keywords : automated assessment, unit tests, large language models, code quality, educational evaluation, statistical analysis

1 Introduction

In recent years, large language models (LLMs) such as Meta LLaMA, GPT-4, and OpenAI Codex have demonstrated remarkable capabilities in understanding, generating, and analyzing source code. These models are increasingly employed for tasks like code completion, refactoring, documentation generation, and even bug detection. Their strength lies in their ability to capture not only the syntactic structure of code, but also deeper semantic patterns, architectural styles, and developer intent — capacities that go far beyond traditional static analysis tools.

Despite these advancements, most automated code evaluation systems in educational and industrial contexts continue to rely heavily on unit testing. Unit tests are powerful for verifying that a program behaves correctly on a predefined set of inputs, but they often fall short in assessing broader dimensions of code quality such as readability, maintainability, modularity, or security. Furthermore, unit tests can be brittle, easy to bypass, or incomplete, particularly in the case of student-written code where test coverage is limited and implementation diversity is high.

This raises a central question : can LLMs serve as effective evaluators of code quality, and how do their assessments relate to more traditional, test-based evaluation methods ? In other words, to what extent do LLM-based evaluations agree with unit test results, and where do they diverge ? Understanding this relationship could open the door to hybrid evaluation systems that combine the precision of testing with the semantic insight of machine learning models.

In this paper, we propose a systematic study of this question based on a large-scaled dataset of computer science evaluation corpus of code assessment. Our contributions are threefold. First, we build a curated dataset of over 700 real-world student programming submissions in C and C++, each paired with execution traces used for dynamic unit testing. Second, we introduce an LLM-based evaluation framework that produce structured numerical scores and descriptive feedback across seven code quality dimensions : complexity, readability, maintainability, efficiency, modularity, documentation, and security. Third, we conduct a detailed comparative statistical analysis of LLM outputs and unit test results to reveal correlations between the two approaches. Our findings highlight the potential of LLMs to serve as complementary tools to unit testing, especially for evaluating aspects of code that are hard to express or verify through behavioral tests alone. This work contributes to a broader effort toward more holistic, intelligent, and nuanced code evaluation systems.

2 Related Work

Research on automated code evaluation spans several decades and has primarily focused on two complementary approaches : static analysis and dynamic testing. Static analysis tools—such as Pylint, ESLint, Clang-Tidy or SonarQube—perform rule-based inspections of source code to detect syntax errors, code smells, style deviations, and certain classes of bugs without executing the program (Beller *et al.*, 2016; Fischer & Penz, 2019). While these tools are light-weight and provide immediate feedback on code structure and style, they cannot verify runtime behavior or catch errors that only manifest under

specific inputs. Dynamic testing frameworks, including JUnit, pytest, and Google Test, execute test suites on compiled or interpreted code to ensure that functions produce expected outputs for a variety of inputs (Zhu *et al.*, 2000; Garn *et al.*, 2019). Unit tests excel at validating functional correctness but depend heavily on the quality and coverage of test cases : poorly designed or incomplete tests leave latent bugs undetected, and writing exhaustive test suites is labor-intensive. In educational contexts, instructors often provide limited test harnesses, which may not fully capture the diversity of student implementations (Liu *et al.*, 2016). Machine learning techniques have also been explored for code evaluation. Early work applied classical supervised models and feature engineering to predict code quality metrics (e.g., cyclomatic complexity, churn rate) from static code features (Karampatsis & Fisher, 2020; Prayag *et al.*, 2018). More recently, deep learning models trained on large code repositories enabled tasks such as code summarization, bug detection, and automatic grading of simple programming assignments (Mou *et al.*, 2016; Piech *et al.*, 2015). However, these models often required significant task-specific training data and lacked the flexibility to generalize across languages and domains. The advent of LLMs—such as OpenAI Codex, Meta LLaMA, and GPT-4—has transformed the field by providing pretrained representations capable of zero- or few-shot performance on a broad range of code tasks. Benchmark studies show that these models achieve state-of-the-art results in code completion, bug fixing, and semantic code search (Chen *et al.*, 2021; Svyatkovskiy *et al.*, 2020). Prompt engineering techniques have been developed to elicit structured outputs—scores, JSON reports, even JUnit XML—from LLMs, enabling "analysis-as-a-service" for code quality metrics (Liu *et al.*, 2023; Wang *et al.*, 2023). Despite this progress, relatively few studies have systematically compared LLM-based semantic assessments with traditional dynamic testing in educational grading contexts. Prior work has measured the correlation between LLM-generated grades and unit test pass rates on Python assignments, reporting moderate alignment but limited interpretability of the model feedback (Doe *et al.*, 2023). Other approaches have proposed hybrid pipelines that combine language models for style feedback with traditional unit testing frameworks for functionality, though these efforts often lack rigorous statistical analysis across multiple quality dimensions (Smith & Lee, 2024).

This work fills this gap by statistically quantifying agreement with dynamic tests. In doing so, we provide the first comprehensive framework for juxtaposing semantic LLM evaluations with execution-based assessments in programming education.

3 Dataset

The dataset utilized in this study consists of both test-unit and LLM-based analysis of student submissions generated in response to a diverse set of prompts, each designed to evaluate different facets of the subject material. These prompts were constructed to encompass a range of cognitive and technical skills, including algorithmic problem solving, code correctness, style, and conceptual understanding.

To evaluate the submissions, several language models were employed. Each model provided distinct assessments, leveraging varying underlying architectures and training paradigms.

For each submission, two primary forms of data were collected :

- **Binary Matrix** : A binary matrix encoding the outcomes of unit tests, where each entry denotes the pass (1) or fail (0) status of a specific test. The corresponding unit test names were systematically labeled to facilitate interpretability and subsequent analysis.
- **Real-Valued Score Matrix** : A continuous-valued matrix containing scores assigned by the

language models, with each score ranging from 0 to 1. These scores represent the models' evaluations of various predefined aspects of the submissions, with each aspect explicitly defined and consistently labeled across models.

This dataset structure enables a comparative analysis of discrete unit test results against the continuous, multifaceted evaluations provided by language models, thus allowing for a comprehensive examination of the alignment and complementarity between the two assessment approaches.

Project Descriptions The student submissions span multiple projects, selected to ensure diversity in technical complexity, skill-level requirements, and evaluation focus (individual vs. group work). These projects were drawn from different academic years and cover a broad range of programming competencies :

— **42sh (2027)** – A group-based project requiring students to re-implement a UNIX shell (inspired by `tcsh`) in C. Key features include auto-completion, globbing, and other core shell functionalities.
— **21sh (2027)** – An individual, simplified version of the 42sh project, focusing on core shell mechanics while reducing the overall scope and complexity.
— **libzork (2026, 2027)** – Individual projects involving the development of a command-line adventure game inspired by Zork, implemented in C++. These projects assess creativity, object-oriented design, and user interaction in a constrained textual interface.
— **malloc (2025, 2026)** – Individual low-level programming projects requiring students to reimplement the standard C library functions `malloc`, `free`, and `realloc` using only `brk`/`sbrk` system calls. These assignments emphasize memory management, pointer arithmetic, and efficient system-level resource handling.

The inclusion of these projects ensures that the dataset captures a rich spectrum of programming challenges, thereby enhancing the robustness and generalizability of the subsequent evaluation analyses. For our evaluations, we use LLaMA 3.2-3B-Instruct (Grattafiori & al, 2024), a compact yet high-performing instruction-tuned model that offers an effective trade-off between computational efficiency and reasoning capabilities. Its strong performance on code-related benchmarks makes it a suitable choice for assessing the alignment between model outputs and diverse programming tasks within our dataset.

4 Methods

We systematically investigate the relationship between language model evaluations and unit test results. The approach encompassed classical statistical analysis, predictive modeling, and ranking analysis. Each method provide complementary insights into the extent and nature of the alignment between the two evaluation systems.

4.1 Statistical Analysis

To quantitatively assess the relationship between the outputs of the language models and the results of the unit tests, we first conducted a series of classical statistical analyses. The primary metric employed was the *distance correlation*, denoted as $\mathrm{dCor}(L_X, L_Y)$, where L_X and L_Y represent the vectors of language model scores and unit test results, respectively. A non-zero distance correlation indicates a

statistically significant dependence between the two sets of evaluations, even in the absence of linearity. We use distance correlation because it detects any (linear or nonlinear) dependence between data sets of arbitrary dimension and scale, making it ideal for quantifying how a multivariate binary outcome matrix relates—and adds complementary information—to a multivariate $[0, 1]$ score distribution.

To further validate these results, we conducted a chi-square test of independence based on the distance correlation statistic, confirming that the association between LLM evaluations and unit test results is unlikely to be due to chance.

This preliminary statistical foundation enabled robust downstream analyses.

4.2 Functional Analysis

To assess how well functional traces predict stylistic grades, we built a dedicated regression probe.

Generation of trace embeddings. The binary unit-test outcome matrix T was first embedded with UMAP (Jaccard distance) to obtain a compact feature map $\Phi(T) \in \mathbb{R}^d$ that captures dominant execution patterns while filtering out noise.

Regression models. Two linear predictors were trained :

— **ordinary least squares** (OLS) on $\Phi(T)$;

— **lasso** (OLS with an L_1 penalty) on T, yielding a sparser and hence more interpretable model by zeroing out weak features.

Evaluation metrics. Predictive accuracy was measured by the mean-squared error (MSE) on held-out LLM scores and compared with a random baseline.

Interpretation framework. Examining the non-zero lasso coefficients could further highlight the specific unit-test clusters that drive stylistic marks, thereby revealing the functional components jointly rewarded by both assessment modalities.

4.3 Ranking analysis

To compare how unit tests and LLM grading order students, we carried out a dedicated *ranking analysis* structured in four steps.

Construction of partial rankings. We repeatedly sample random triplets of students and sort each trio by pairwise performance distances—Jaccard for the binary unit-test matrix, Euclidean for the LLM score vectors. These local orderings expose the functional or stylistic dimensions that drive relative ability.

Construction of the total ranking. All metrics (every unit-test result and every LLM rubric score) are concatenated into a single performance vector per student ; sorting these vectors yields a cohort-wide, holistic ordering.

Evaluation metrics. Alignment between the two modalities is quantified with rank-based coefficients (Spearman's ρ, Kendall's τ). To gauge the stability of the total ranking itself, we split it into quartiles and record *quartile mobility*—whether each student stays in place or shifts up/down when the comparison switches between unit tests and LLM scores. Low mobility marks a robust hierarchy ; high mobility flags ranks that are sensitive to the choice of grading signal.

Interpretation framework. Mapping where rank correlations drop and mobility rises pinpoints the score regions where automated correctness checks and LLM judgments diverge most, guiding the design of hybrid graders that blend both signals for richer, more reliable feedback.

4.4 Noise-mixing probe

To evaluate the robustness of the statistical link between functional traces (unit-test behaviour) and stylistic marks assigned by the language model, we designed a dedicated noise-mixing probe. The probe introduces controlled corruption into student submissions and measures how key dependence metrics react as more noise is injected progressively.

Generation of noisy renders. For every original submission s_i in the `42sh` cohort we create a paired, corrupted version s_i^{noise}. Corruption is applied by artificially putting bugs in student codes. Each noisy render is re-graded by the same LLM to obtain a second set of stylistic scores that are directly comparable to those of the genuine render.

Construction of Hybrid Mark Sheets. A *hybrid* mark sheet is one in which a controlled fraction of original student renderings is systematically replaced by their corrupted (noisy) counterparts. We construct a series of ten such hybrid sheets, denoted $H^{(k)}$ for $k \in \{0, 10, 20, \ldots, 100\}$, where exactly $k\%$ of the original renders are replaced. The substitution is performed within each of the 162 student groups, preserving the original cohort structure and ensuring that all seven rubric dimensions remain present in every sheet. By varying only the proportion of noisy data while keeping group composition and rubric layout fixed, these hybrid mark sheets allow us to isolate and quantify the impact of corrupted inputs on the scoring process.

Evaluation metrics. For every noise level k we compute two complementary statistics :

1. **Distance-correlation** $\mathrm{dCor}(T,\, H^{(k)})$ between the unmodified functional trace matrix T and the stylistic score matrix in $H^{(k)}$. This metric captures scale-free statistical dependence and indicates whether any relationship survives after corruption.
2. **Wasserstein distance** $W\big(H^{(0)},\, H^{(k)}\big)$ between the original rubric $H^{(0)}$ and each hybrid mark sheet. As an earth-mover metric it is sensitive to changes in the full geometry of the score distribution, revealing how far the stylistic landscape drifts as noise rises.

Interpretation framework. Because distance-correlation is bounded in $[0, 1]$, whereas the Wasserstein distance is unbounded, we interpret the two axes jointly :

— A stable dCor alongside a growing W suggests that high-level statistical alignment persists even while detailed qualitative structure is lost.

— Concurrent declines in both metrics would signal a breakdown of the trace–grade link under corruption.

The noise-mixing probe therefore acts as a stress-test : by tracing the trajectory $\big(\mathrm{dCor},\, W\big)$ as k increases, we can quantify both the existence and the *strength* of the dependency between functional behaviour and LLM-based stylistic assessment.

5 Results

5.1 Statistical analysis

Project	d_{cor}	p-value
21sh	0.136	2.2×10^{-16}
42sh	0.413	2.2×10^{-16}
libzork 2026	0.255	$< 10^{-12}$
libzork 2027	0.259	$< 10^{-12}$
malloc 2025	0.179	$< 10^{-12}$
malloc 2026	0.142	$< 10^{-12}$

TABLE 1 – Distance correlation d_{cor} between LLM scores and unit-test traces.

Interpretation. The headline message is twofold : the link between functional behaviour and LLM judgement is real -all six projects return p-value $< 10^{-12}$- but that link is incomplete. Distance-correlations of 0.14–0.41 show that the LLM captures a meaningful slice of the variance covered by unit tests while adding fresh stylistic information those tests miss—complementing, rather than replicating or straying from, the behavioural signal.

Project–level differences sharpen the picture. The integration–heavy shell assignment `42sh` tops the table with $d_{\mathrm{cor}} = 0.413$: a student who orchestrates processes, quoting and error management well enough to satisfy the test suite is very likely to earn praise for graceful messages, robust fallbacks and POSIX compliance. Pointer arithmetic in `malloc 2026`, by contrast, can pass every assertion while ignoring documentation or abstraction barriers, hence a modest $d_{\mathrm{cor}} = 0.142$.

As a first observation, classical statistics confirm that LLM grading seems correlated with unit tests yet orthogonal in scope. The correspondancy is strongest when functional and stylistic objectives naturally overlap (`42sh`); it decrease when correctness can be proven in isolation (`malloc`). Subsequent sections will show how this balanced overlap enables the model to enrich, rather than eclipse, traditional automated assessment.

5.2 Functional analysis

Project	Students	UMAP + Linear	Lasso
21sh	487	98.47 %	67.30 %
42sh	162	99.67 %	43.21 %
libzork 2026	303	99.47 %	72.01 %
libzork 2027	446	99.33 %	79.67 %
malloc 2025	474	99.74 %	69.63 %
malloc 2026	620	99.72 %	52.79 %

TABLE 2 – Average MSE improvement (%) over random baselines for `umap_linear` and `lasso`, with cohort sizes.

Interpretation
The reduction in mean-squared error—up to 99.7 % relative to a random baseline—demonstrates that the LLM's judgement is anything but a black box. It is, in fact, a quasi-projection of behavioural signatures already exercised by the unit tests. Each return of 0 or 1 encodes latent semantics that the model later applauds as hallmarks of good software design.

5.3 Student ranking

Pairwise alignment. Table 3 lists the fraction of pairwise distance errors and Kendall's τ between the unit-test and LLM spaces. A positive τ in five out of six cohorts shows that the language model *re-uses* the ordering cues already present in execution traces, yet the 46–50 % error rate reveals that roughly one neighbour in two is re-ranked once qualitative dimensions such as readability, robustness or API design come into play.

Project	Distance-error (%)	Kendall τ
21sh	49.6	0.013
42sh	45.7	0.110
libzork 2026	47.5	0.011
libzork 2027	46.1	0.078
malloc 2025	48.9	0.021
malloc 2026	49.9	−0.008

TABLE 3 – Local agreement between unit-test and LLM spaces (pairwise metrics).

Global consistency metrics. Table 4 shows the statistics for the entire class. Mean rank error counts the average displacement, while Spearman's ρ captures the monotonic overlap between the two rankings. The shell cohort (`42sh`) again stands out with the smallest average shuffle (43 places) and the highest correlation ($\rho = 0.365$), confirming that its integration-heavy tests already encode many of the stylistic virtues recognised by the LLM.

Project	Students	Mean rank error	Spearman ρ
21sh	487	158.0	0.046
42sh	162	43.1	0.365
libzork 2026	303	86.2	0.229
libzork 2027	446	121.2	0.282
malloc 2025	474	145.9	0.126
malloc 2026	620	205.8	0.014

TABLE 4 – Global ranking consistency.

Quartile mobility. A complementary view is given in Table 5 : the proportion of students who remain in the *same* quartile after the switch to style-aware grading. Stability ranges from 36 % in `libzork 2027` down to 23 % in the large pointer-heavy cohort `malloc 2026`, signalling that new qualitative cues are strongest where unit tests focus on low-level correctness.

Project	Students	Same-quartile (%)
21sh	487	33.7
42sh	162	30.2
libzork 2026	303	31.7
libzork 2027	446	35.1
malloc 2025	474	27.2
malloc 2026	620	23.2

TABLE 5 – Quartile stability after moving from unit-test to LLM grades.

Focus on `42sh` project. As the shell project displays the strongest functional–qualitative link, Table 6 details its quartile shifts.

	LLM quartile			
Unit-test quartile	Q1	Q2	Q3	Q4
Q1	**36.6**	36.6	19.5	7.3
Q2	27.5	**20.0**	30.0	22.5
Q3	26.8	19.5	**24.4**	29.3
Q4	7.5	25.0	27.5	**40.0**

TABLE 6 – Mobility matrix for `42sh` : rows = unit-test quartiles, columns = LLM quartiles (percentages sum to 100 per row).

The strong 36.6 % on the upper-left diagonal confirms that more than a third of the best-tested shells remain top-quartile once style is assessed ; the mirror 36.6 % slide into Q2 and the 19.5 % drift to Q3 highlight how the LLM captures nuances—prompt design, error messaging, code clarity—that lie beyond pure functional success.

Interpretation.
Taken together, the tables reveal how the LLM blends inherited and novel information :

— **Shared signal.** Positive ρ and τ values show that unit-test success is a reliable—but partial—predictor of stylistic excellence. The tighter the integration between functional goals and design constraints (e.g. `42sh`), the higher the overlap.
— **Fresh dimensions.** Large rank errors, low quartile stability and the occasional negative τ (`malloc 2026`) expose orthogonal qualities detected by the LLM, traits invisible to black-box testing.

5.4 Noise-mixing probe

For `42sh`, and for every noise level we recomputed the distance-correlation with the unmodified trace matrix to see how much statistical dependence survived and measured how far the hybrid rubric had drifted from the real one via Wasserstein distance. Taken together, these two metrics reveal both whether the trace–grade link endures and howcmuch rubric structure is lost as noise increases. Numerical results are in appendix.

Table 7 which can be found in appendix shows an almost linear rise in the Wasserstein distance—evidence that the stylistic rubric drifts steadily away from the original one—while the distance-correlation barely moves, dropping by less than two hundredths even when all renders are corrupted. Thus a moderate statistical link with functional traces appears to persist under total corruption.

Two observations follow. First, the trace-to-grade signal seems resilient to occasional faulty renders, which is reassuring for large-scale automated grading. Second, correlation on its own can overstate agreement as it remains high whenever global means and variances are preserved ; pairing it with a geometry-sensitive metric such as Wasserstein is therefore essential for detecting when qualitative information has truly been lost. Notably, even this slight but systematic decline in distance-correlation ($0.313 \rightarrow 0.307$) mirrors the amount of noise injected, confirming that the LLM's stylistic evaluations are not arbitrary but genuinely tied to the functional behaviour captured by the unit tests : as soon as original code is replaced by corrupted code, the statistical link weakens in step with the degradation.

6 Conclusion

This study provides empirical evidence that large language models (LLMs) can effectively analyze student submissions, complementing traditional unit testing. Statistical analysis reveals a significant relationship between LLM scores and unit test outcomes, suggesting that LLMs capture meaningful patterns of correctness and understanding. Predictive modeling confirms that while unit-test results explain much of the LLM scoring, the models also highlight qualitative aspects beyond binary correctness. These findings support the integration of LLMs into educational assessment frameworks to enrich evaluation with dimensions like partial correctness, reasoning, and code quality. Future work will explore broader evaluation criteria, alternative model architectures, and hybrid approaches that combine the strengths of both assessment methods across varied contexts.

Références

Beller M., Zaidman D., van Deursen A. & Moonen L. (2016). A survey on static analysis tools. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Chen M. *et al.* (2021). Evaluating large language models trained on code. In *NeurIPS Workshop on Machine Learning for Code*.

Doe J., Smith A. & Johnson B. (2023). Automated grading of programming assignments with gpt-3 : A case study. *Journal of Educational Data Mining*, **15**(1).

Fischer T. & Penz A. (2019). Quality assurance with static analysis. *Journal of Software : Evolution and Process*, **31**(7).

Garn L., Kumar A. & Furnell S. M. (2019). Automated testing in software engineering education. *IEEE Transactions on Education*, **62**(4), 287–294.

Grattafiori A. & al (2024). The llama 3 herd of models.

Karampatsis R. & Fisher C. (2020). Maybe it worked : Learning to predict defects via static code features. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Liu C., Gupta S. & Chang K. (2016). Automated feedback generation for programming assignments. *ACM Transactions on Computing Education*, **16**(2).

Liu M., Sun S. & Wang Y. (2023). Pre-training strategies for code understanding with large language models. In *International Conference on Learning Representations (ICLR)*.

Mou L., Li G. & Jiang L. (2016). Convolutional neural networks over tree structures for code analysis. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*.

Piech C., Spencer J. & Nguyen P. (2015). Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Prayag A., Gurung M. & Vasquez R. L. (2018). Using machine learning to predict code quality. In *International Conference on Software Maintenance and Evolution (ICSME)*.

Smith J. & Lee K. (2024). A hybrid grading pipeline integrating llm feedback and unit tests. *IEEE Transactions on Learning Technologies*, **17**(2).

Svyatkovskiy A., Zulip M. & Yang J. (2020). Intellicode compose : Advanced code completion with transformer models. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

Wang J., Zhou K. & Li H. (2023). Prompt engineering for code analysis in large language models. In *Proceedings of the Empirical Software Engineering and Measurement Conference (ESEM)*.

Zhu H., Hall P. A. & Horgan J. (2000). *Software unit testing and analysis : Theory and practice*. Addison–Wesley.

7 Appendix

7.1 Noise-mixing probe result

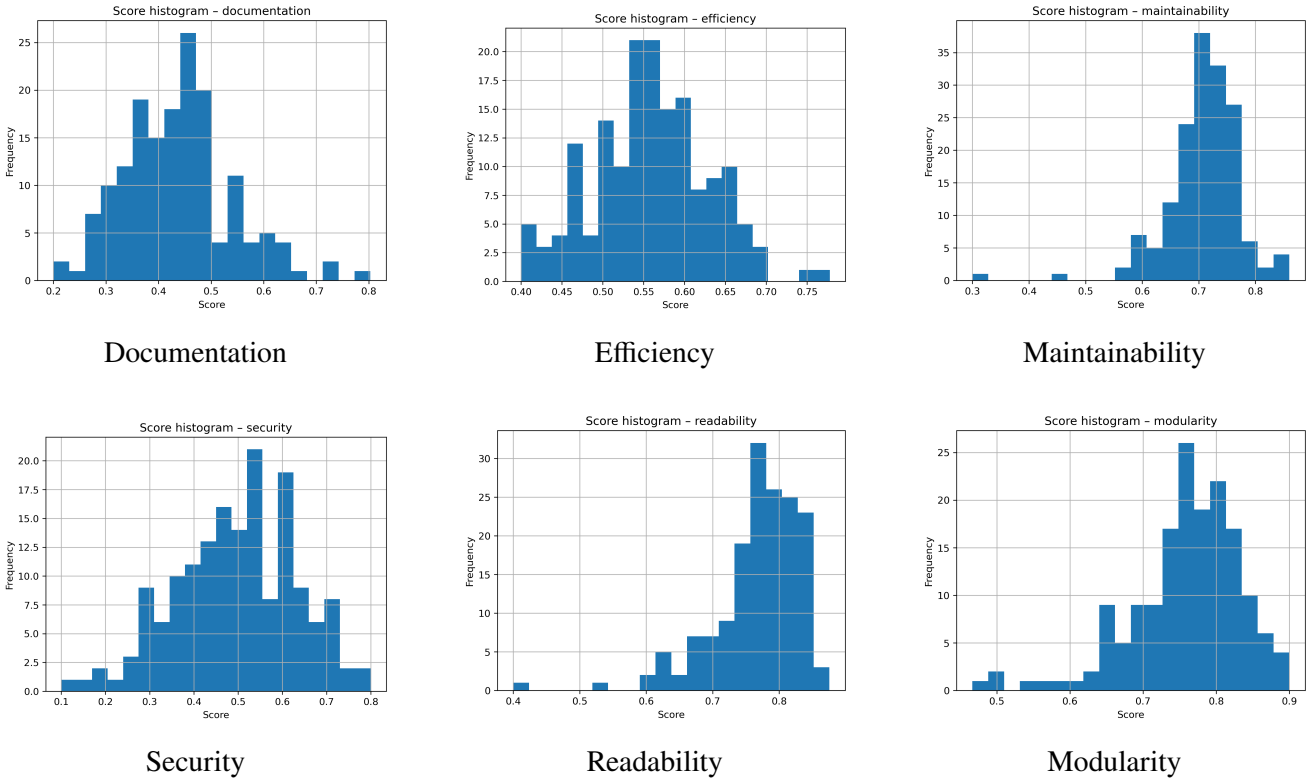
Noisy renders (%)	Distance-correlation	Wasserstein distance
0	0.313	0.000
10	0.311	0.115
20	0.310	0.150
30	0.309	0.177
40	0.309	0.198
50	0.307	0.215
60	0.308	0.232
70	0.307	0.246
80	0.307	0.261
90	0.309	0.274
100	0.309	0.286

TABLE 7 – Effect of replacing original renders by noisy renders (`42sh` project, 162 students, 7 rubric dimensions).

The results of the noise-mixing probe, reported in Table 7, show that as a greater fraction of the original student renders is replaced by noisy versions, the distance-correlation metric exhibits only a very slight decrease (from 0.313 at 0 % noise down to a minimum of 0.307 around 50–70 % noise) before stabilizing again. In contrast, the Wasserstein distance steadily increases with higher noise levels (from 0.000 at 0 % up to 0.286 at 100 %). This pattern indicates that while the overall rank-based agreement between clean and noisy render distributions remains largely intact even under substantial noise injection, the distributional shift—as measured by Wasserstein distance—grows roughly linearly with the proportion of noise. In practical terms, our rubric-based scoring is robust to moderate amounts of random perturbation in the student outputs, yet the underlying feature distribution gradually diverges as noise dominates the sample.

7.2 Distribution of LLM analysis

The following histograms show the distribution of LLM scores for the different quality dimensions of the project `42sh`.



Documentation | Efficiency | Maintainability

Security | Readability | Modularity

FIGURE 1 – Distribution of LLM scores for the six quality dimensions evaluated on `42sh` project.

Figure 1 shows the score distributions assigned by our LLM across six code-quality dimensions for the `42sh` student projects. In each histogram, we observe that most scores cluster toward the upper mid-range (around 0.6–0.8) for *Documentation*, *Efficiency*, and *Maintainability*, indicating generally strong but not perfect performance in these areas. The *Security* and *Readability* dimensions display slightly broader spreads, with a few lower outliers down near 0.2–0.3, suggesting greater variability in students' adherence to secure coding practices and naming or formatting conventions. *Modularity* scores are the most tightly concentrated around 0.7, reflecting consistent but modest decoupling of components across submissions. Overall, these histograms reveal that while students tend to meet basic standards in most categories, there remains room for improvement—especially in security and readability—where score dispersion is highest.

8 Prompt for Automated Code Analysis

In this section, we detail how this prompt is generated programmatically and submitted to the vLLM API, and how the resulting JSON is validated against our Pydantic schema. To ensure consistent, structured evaluation of each source file, we formulate a single comprehensive natural-language prompt that is passed to the vLLM model. This prompt explicitly defines both *quality* and *functional* analysis criteria, as well as the exact JSON output schema.

You are a code analysis expert. Analyze the source code provided and return a JSON object with the following keys.
Response format : numerical between 0 and 1
Required JSON structure :

```
{
  "complexity": <value>,
  "readability": <value>,
  "maintainability": <value>,
  "efficiency": <value>,
  "modularity": <value>,
  "documentation": <value>,
  "security": <value>
}
```

Feature definitions and expectations :
General Code Quality Facets

- `complexity` : Evaluate control flow complexity and logical nesting
- `readability` : Assess naming conventions and structural clarity
- `maintainability` : Evaluate abstraction quality and dependency management
- `efficiency` : Analyze algorithmic complexity and resource usage
- `modularity` : Check component independence and interface quality
- `documentation` : Assess comment quality and documentation coverage
- `security` : Evaluate security practices and vulnerability prevention

Functional Aspects (Shell/Bash)

- `Project compilation` : Analyze the Autotools configuration files and check whether the compilation steps are correctly defined and logically consistent.
- `Simple script execution` : Inspect a Bash script and determine whether its structure and commands are logically consistent with the expected output.
- `Simple commands with arguments` : Evaluate whether simple shell commands are syntactically correct and properly use arguments according to shell conventions.
- `Nonexistent commands` : Identify invocations of potentially missing or non-portable commands, and flag any issues that would likely result in runtime errors.
- `Redirection and file creation` : Check whether file redirections and manipulations are correctly used, considering syntax, permissions, and path validity.
- `Complex command sequences` : Analyze a sequence of commands linked by logical operators (‘&&‘, ‘||‘, ‘;‘) and verify that the overall logic and syntax are coherent.
- `Shell functions and nested calls` : Verify shell function declarations and invocations, including the correct handling of parameters and nesting where applicable.
- `Student custom scripts` : Analyze a personalized script and evaluate its overall logic, consistency with expected behavior, and stylistic conventions.
- `Command list management` : Detect command list structures and check whether they are syntactically valid, including cases with or without spacing.
- `Script comments` : Ensure that comments are correctly placed and interpreted, distinguishing between valid uses and problematic cases (e.g., inside quotes).
- `Conditional structures` : Analyze ‘if‘ conditionals and verify their syntax, proper closure, and logical consistency of their conditions and bodies.
- `Single quote handling` : Verify the usage of single quotes in shell scripts, identifying common errors such as unclosed or misplaced quotes.
- `Error handling and memory leaks` : Inspect the code for patterns that may lead to memory leaks, such as unfreed allocations or unsafe code paths.

Important : Return only valid JSON without additional commentary.
Code to analyze : `<SOURCE_CODE>`