One Workflow to Grade Them All: LLM-Enabled Auto-grading for Comprehensive, Personalized Feedback

Mohit Chandarana, Codio

Mohit has a BE in Computer Engineering and an MS in Computer Science. From generating insightful learning analytics for CS Educators to prototyping novel product features and algorithms, he works towards bridging the gap between cutting-edge academic research and its application in the industry in his role at Codio.

Sindhu Ramachandra, Codio

A passionate Data Science professional with a strong foundation in data analytics, large language models, MCP servers and prompt engineering, currently expanding expertise at Codio.

One Workflow to Grade Them All: LLM-Enabled Auto-grading for Comprehensive, Personalized Feedback

Abstract

Amid rapid industry change, new AI tools and the growing popularity of "vibe-coding" applications are fueling interest in learning computer science across a wide range of instructional settings. This growing demand may intensify an existing challenge in computing education: heavily skewed instructor-to-learner ratios that make it difficult to provide timely formative feedback on assessments and projects. Autograding tools help address this problem, but even strong autograders often require substantial manual configuration and typically provide limited insight into the underlying reasons for test-case failures or the misconceptions reflected in student submissions. Because learners often turn to LLM-based tools to understand why something failed, this creates an opportunity to integrate them into assessment workflows in ways that better support learning.

This paper analyzes an LLM-based grading workflow using thousands of historical student submissions from 70 Python code-writing exercises in an asynchronous MOOC-style course on Codio. The assessments span a wide range of autograder complexity, including simple input-output tests, unit-test-based checks, and custom autograders that parse student submissions.

Our analysis shows that, for this dataset, the LLM-based workflow achieves high agreement with the existing auto-graders across a wide variety of test configurations, while simultaneously producing fine-grained, test-case-level explanations for students. We illustrate how this workflow can surface rubric-like feedback from existing auto-graders with minimal additional configuration, and we discuss practical opportunities and limitations for integrating LLM-generated grading and feedback into CS courses.

Introduction

As Generative AI becomes embedded across industries, demand for computing skills continues to grow. Computing programs have seen sustained increases in interest and enrollment, with many institutions reporting that computer science is now one of the largest undergraduate majors [23]. Although recent survey data suggests undergraduate computing enrollments may have stabilized in 2025–26 [26], labor projections still anticipate "much faster than average" growth in computer and information technology occupations from 2024 to 2034 [24].

This growth intensifies a long-standing challenge in computing education: providing timely, explanatory feedback on programming assignments at scale. Instructors often face heavily skewed instructor-to-learner ratios, limiting the individual support they can offer during labs, office hours, and assignment review. Autograding tools have become essential for managing

large cohorts because they can run tests over student code and return immediate pass/fail results. However, most such tools focus primarily on correctness, with limited support for explaining why a submission fails, how it relates to rubric criteria or common misconceptions [8,11,14,22]. Richer feedback typically requires substantial additional configuration or separate rubric-focused mechanisms.

At the same time, Large Language Models (LLMs) have opened new possibilities for programming feedback. Prior work has explored LLMs for generating programming exercises [19], improving error messages [10], explaining code and concepts [9, 17], generating unit tests [1, 20, 21, 25], and grading code against specified criteria [2, 12, 16]. In practice, many students already use general-purpose LLM tools to interpret failing tests and error messages [3, 18]. Yet many of these approaches either require instructors to author new, question-specific rubrics and prompts or operate outside existing autograding workflows, making them harder to integrate into established assessment pipelines.

In this work, we investigate a complementary approach: using LLMs to leverage existing autograding infrastructure rather than replace it. We design and evaluate an LLM-enabled workflow that takes as input the autograding artifacts already used in a course, such as input-output tests, unit test suites, or custom checking scripts, and asks an LLM to (a) derive human-readable, rubric-like criteria that describe what is being tested and (b) use those criteria to grade student submissions and generate test-case-level explanations. We apply this workflow to thousands of historical submissions from 70 Python coding exercises in an asynchronous MOOC-style course, spanning autograders from simple checks to scripts that inspect code structure.

Our analysis is guided by three research questions:

RQ1: Rubric reconstruction. To what extent can we generate accurate, rubric-style criteria from existing autograding scripts, test cases, or instructions, and establish a reliable mapping between these criteria and the original autograder elements?

RQ2: Grading alignment. How accurately can an LLM grader, using the generated criteria, reproduce the decisions of the existing autograder across a large set of historical submissions?

RQ3: Feedback characteristics. What kinds of test-case-level feedback does this workflow produce for passing and failing submissions? How does this feedback align with the structure and intent of the original autograder?

By examining how LLMs can surface rubric-like information from existing autograders and align grading decisions with existing tests, this study provides practical insight into how instructors might scale richer, more structured feedback using assessment infrastructure they already have in place.

Background and Previous Work

Test-driven autograding platforms and workflows

Test-driven autograding has become a core component of programming instruction at scale. In a typical workflow, instructors provide unit tests or executable checks, and the autograder runs these checks in a controlled environment to return pass/fail outcomes, scores, and (sometimes) error traces [8], [11], [14]. Large-scale platforms and course tooling increasingly standardize the execution environment and submission pipeline, enabling rapid feedback cycles and repeated submissions. These features are particularly valuable in introductory computing courses.

A number of widely used systems illustrate the strengths of this approach. **Otter-Grader** supports grading Python and R code across scripts, notebooks, and related formats [27]. **nbgrader** provides an end-to-end workflow for notebook-based assignment authoring and grading, combining autograded cells with instructor-facing tools for managing submissions [28]. Also, there are platforms that execute an instructor-provided autograder and support hybrid workflows where automated checks can be paired with manual rubric scoring when needed [31]. Related systems also support online assessments through parameterized problem templates and instructor-defined grading backends [29].

Across these systems, a common thread is that the **instructor's tests and scripts define the grading logic**. As a result, the dominant feedback paradigm remains test-centered: learners see which tests passed or failed, along with outputs or runtime diagnostics. While this workflow can provide immediate, objective feedback, it typically does not *by itself* produce pedagogically organized rubric criteria (e.g., mapping test outcomes to conceptual learning objectives) or narrative explanations that connect failures to likely misconceptions. In practice, instructors often need to manually construct rubrics, author feedback messages, and maintain mappings between tests and rubric items, especially when assignments evolve over time or when test suites were inherited from prior offerings [8], [11], [14].

These limitations also appear in systematic reviews of programming feedback tools. Messer et al. report that many tools emphasize correctness via dynamic testing and that feedback is often limited to pass/fail outcomes and output differences, with fewer tools addressing deeper explanation or higher-level qualities such as readability and maintainability [11]. Keuning et al. similarly characterize automated feedback as frequently focused on identifying failures rather than supporting conceptual repair strategies [8].

Beyond pass/fail: richer feedback in automated assessment

Prior work has explored several approaches that move beyond correctness alone and instead connect student errors to conceptual understanding or generate more actionable feedback. ConceptGrader, for example, evaluates student submissions in terms of conceptual correctness rather than only final output, offering a pathway toward feedback that is closer to rubric-based assessment [4]. Other work uses program repair to generate actionable feedback by proposing

minimal changes that would correct a submission [15]. Haldeman et al. demonstrate an approach in which an ML classifier links incorrect submissions to pre-written instructor hints targeting likely misconceptions [6]. Additional work has also shown that how formative feedback is delivered can matter as much as what is delivered [7].

These approaches demonstrate that it is possible to deliver more meaningful feedback than pass/fail test output. However, many require significant upfront investment: building concept models, curating misconception labels, designing repair strategies, or authoring feedback libraries and mapping rules [6], [15]. Moreover, even where advanced feedback is available, instructors may still face the practical challenge of translating existing autograder suites into a form that yields consistent, rubric-like feedback at scale.

LLMs for programming assessment and feedback

Large language models (LLMs) have introduced new opportunities for producing explanatory feedback and supporting debugging and conceptual understanding in programming contexts [3], [18]. Prior work has examined LLMs for explaining code, improving error messages, acting as tutors that generate natural-language guidance [9], [10], generating programming exercises [19], and generating unit tests [1], [20], [21], [25]. Early empirical work and case studies have also investigated the feasibility of using LLMs to grade code against specified rubrics, highlighting both potential value and concerns about calibration, reproducibility, and reliability [2], [12], [16].

FlexiGrader represents a related direction: an LLM-based autograder for flexible, open-ended CS1 assessments, with an emphasis on narrative, personalized feedback for diverse student projects [30]. This line of work highlights the promise of LLM-based grading, but it also underscores persistent challenges, including consistent grading across diverse submissions, limiting hallucinations, and reducing instructor overhead due to prompt design and calibration.

Our work sits at this intersection, but with a distinct emphasis: rather than replacing existing autograders or requiring instructors to author new rubrics from scratch, we evaluate an LLM-enabled workflow that **leverages the grading logic already encoded in test suites and scripts**, and translates that logic into structured, human-readable criteria to support consistent, test-case-level explanations. In doing so, we target a practical bottleneck highlighted across both platform practice and research: the gap between *what tests check* and *how instructors want to communicate that evaluation to learners*.

Methods

Dataset and assessment selection

We collected historical assessment data from an asynchronous MOOC-style introductory Python course on the Codio platform between July 2024 and June 2025, following IRB-approved anonymization and consent procedures. The course was organized into multiple modules, each containing several assignments. For this study, we focused on the code-writing knowledge-check

assessments used in the final assignment of each module. These assessments contained the executable autograding artifacts analyzed in our workflow.

Across all the selected assessments, the autograding artifacts took three main forms: input/output test cases, Python **unittest** test suites, or custom scripts that parsed or inspected student code for required behavior or features. These assessments were authored by multiple instructors and yielded two linked datasets: **assessmentDetails**, containing assessment metadata and autograder artifacts, and **assessmentAttempts**, containing submission-level records for each learner attempt. The initial dataset contained **290,000** assessment attempts from **5,454** unique learners across **70** assessments.

Category	**Assessment-level data** *(assessmentDetails)*	**Attempt-level data** *(assessmentAttempts)*	**Used For**
Identifiers	assessmentId, assignmentName, moduleName	assessmentId, attemptId, assignmentId, submittedByUserId, submittedAtTimestamp	Linking Records
Task Content	assessmentType, questionText, sampleSolution	submissionFileContent	Rubric Generation, Grading
Autograder Artifacts	autograderType, autograderScript, inputOutputTestCases	runResult (exit code, stderr, stdout), points	Rubric Reconstruction, Validation/Evaluation

Table 1: Assessment Details and Attempts Data points

Pre-processing and Sampling

Because introductory/CS 1 level exercises often yield many repeated or near-identical submissions with the same observable execution behavior, we first deduplicated the **attempts** dataset. To do so, we retained rows with unique combinations of **submissionFileContent**, **stderr**, **exit code**, and **stdout**. This step reduced redundant submissions while preserving variation in student code and autograder results. After deduplication, we randomly sampled 200 submissions per assessment, with a balanced distribution of passing and failing attempts, to optimize for LLM compute costs. This sampling step reduced the dataset to a total of **14,000** submissions across all selected assessments.

Workflow Overview

We designed the workflow in two stages: first, reconstructing rubric-style criteria from existing autograder artifacts, and second, using those criteria to grade student submissions and generate feedback. Separating these stages improved interpretability, while allowing each stage to be

analyzed independently. Both stages were implemented as Python pipelines that ingested course data, executed batch API calls, and stored structured outputs for analysis. We used the Claude Sonnet 3.7 model (Anthropic) for all experiments and did not perform a comparative evaluation across model families or providers. Across both stages, we used a one-shot prompting setup with structured XML prompts and one fully worked example to standardize outputs. This design allowed us to evaluate whether a simple prompt framework could generalize across diverse assessment and autograder contexts.

Stage 1: Rubric reconstruction and validation

In the first stage, we prompted an LLM to reconstruct rubric-style criteria from the grading logic already encoded in the autograder artifacts. For each assessment, the prompt included the available task context and grading materials, including the question instructions, and sample solution. The model was asked to identify each distinct test or validation block represented in the autograder and convert it into a corresponding human-readable rubric item. To preserve the granularity of the original grading logic, the prompt instructed the model to maintain a one-to-one mapping between detected checks and generated rubric criteria. Each generated rubric item was returned in a structured format containing a rubric ID, a short title, and a description of the grading criteria.

To support systematic validation, we parsed the original autograder artifacts into a structured comparison dataset with one row per assessment. For each assessment, this dataset recorded the autograder type, the number of distinct checks identified in the autograder logic, and each individual check as a separate structured entry. We then constructed a parallel dataset for the reconstructed rubrics, recording the number of generated rubric items and the content of each rubric criterion in the same format.

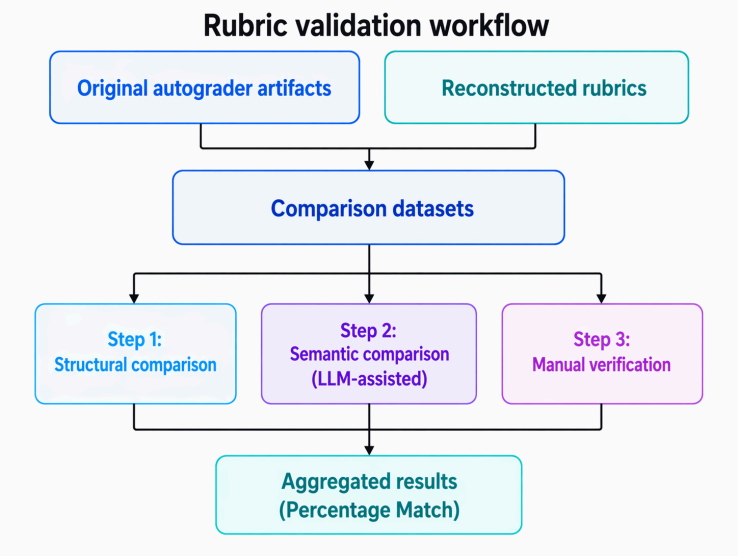

Figure 1: Validation workflow for mapping generated rubric items to original autograder checks.

Validation proceeded in two steps. First, a Python script compared the autograder and rubric datasets for each assessment on structural dimensions, including the number of distinct checks identified and their relative ordering, where ordering was used only as a consistency check on output structure rather than as evidence of semantic correctness. Second, we evaluated whether each generated rubric item matched the functionality of the corresponding autograder check through semantic comparison, supported by LLM-assisted matching and manual verification.

Stage 2: Criterion-level grading and feedback

In the second stage, we used the reconstructed rubric criteria to evaluate historical student submissions at the criterion level. For each submission, the prompt included the question instructions, sample solution, student submission, and the reconstructed rubric criteria. The model was instructed to analyze the assessment task, reference solution, and student code in relation to the rubric, and then evaluate each rubric criterion independently. It returned a structured output containing the rubric ID, rubric title, a binary judgment (**CORRECT** or **INCORRECT**), and criterion-level feedback. For criteria judged incorrect, the model was instructed to produce brief, actionable feedback explaining what was wrong and how the submission could be improved.

To evaluate grading alignment, we parsed the model's criterion-level outputs into a structured analysis dataset aligned with the sampled submissions dataset. For each rubric criterion, we compared the model's **CORRECT/INCORRECT** judgment against the corresponding autograder outcome to identify agreements and mismatches. These criterion-level comparisons formed the basis for the alignment metrics reported later. Because this stage also supported our analysis of feedback characteristics, we separately reviewed the generated feedback for failing criteria to examine whether it was specific, actionable, and aligned with the functionality being tested. The prompt templates and supplementary materials for this study are available in the following GitHub repo: github.com/codio-content/asee-2026-one-workflow-to-grade-them-all.

Analysis and Results

Results: RQ1 - Rubric reconstruction fidelity

Can we generate accurate rubric criteria for grading from any given auto-grading script, test-cases or instructions?

Across the 70 assessments, 68 had autograder artifacts that could be exported and parsed for rubric reconstruction. Of these 68 assessments, 67 produced complete one-to-one mappings between reconstructed rubric items and original autograder checks. The remaining assessment exhibited a single unmapped check. Excluding the two export/parsing failures, this corresponds to complete mappings for **98.5%** of assessments. These results indicate that, across the autograder formats represented in this course, the rubric reconstruction stage reliably preserved both the granularity and intent of the underlying checks.

Module Name	# of Assessments	# of Assessments with 100% match	# of mismatched rubric items
Basic Fundamentals	5	5	0
Operators	3	3	0
Conditionals	5	5	0
Loops	2	2	0
Lists	5	5	0
Strings	5	5	0
Files	5	5	0
Functions	5	5	0
Recursion	5	5	0
Objects	5	5	0
Mutability	5	5	0
Encapsulation	5	4	1
Inheritance	5	5	0
Polymorphism	5	4	1
Advanced	5	4	1

Table 2: Summary of rubric reconstruction fidelity by module.

Results: RQ2 - Grading alignment

Can LLMs accurately apply the generated rubrics to evaluate student submissions, correctly identifying passing and failing test cases?

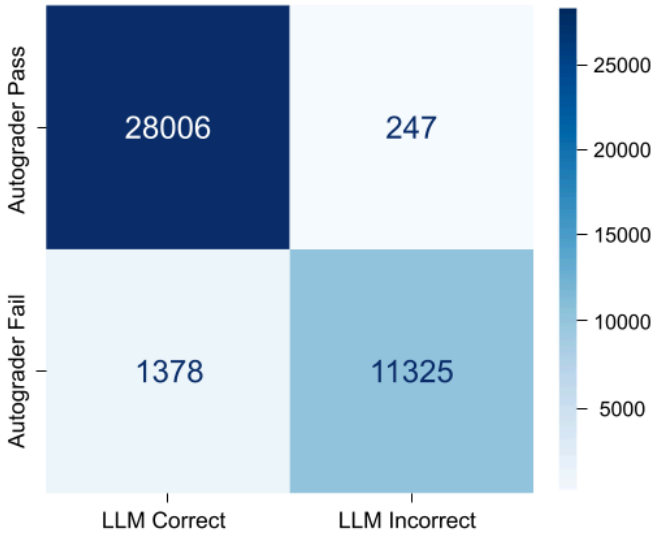

Figure 2: Criterion-level confusion matrix comparing LLM rubric-based grading judgments with corresponding autograder outcomes across all evaluated rubric criteria.

We evaluated grading alignment at the criterion level by comparing the model's **CORRECT/INCORRECT** judgments with the corresponding autograder outcomes across the sampled submissions. In total, this analysis covered 14,000 submissions and 40,956 criterion-level evaluations. As shown in Figure 2, agreement between the LLM grader and the autograder was high overall, with 28,006 true positives and 11,325 true negatives, alongside 247 false negatives and 1,378 false positives.

Accuracy	96%
Precision	95.3%
Recall	99%
F1 score	0.97

Table 3: Accuracy, Precision, Recall and F1 score

Table 3 reports the resulting summary metrics. These results indicate that the LLM grader generally reproduced the autograder's criterion-level pass/fail decisions when operating over the reconstructed rubric criteria. High recall suggests that the model rarely marked autograder-passing criteria as incorrect, while the lower false-positive rate indicates that most model-assigned **CORRECT** judgments were also supported by the original autograder. Together, these results suggest that rubric-based LLM grading can align closely with existing autograder judgments at scale.

Results: RQ3 - Feedback characteristics

What kinds of test-case-level feedback does this workflow produce for passing and failing submissions, and how does this feedback align with the structure and intent of the original autograder?

Beyond criterion-level judgments (**CORRECT/INCORRECT**), the grading workflow produced brief natural-language feedback for each rubric item, intended to correspond to a single underlying autograder check. To examine the character of this feedback, we conducted a qualitative review of student submissions alongside the reconstructed rubric criteria, criterion-level grading decisions, and associated feedback. This review focused on whether the feedback (i) targeted the same behavior as the underlying check, (ii) was specific enough to support debugging, and (iii) preserved the one-check-per-feedback-unit granularity implied by the original autograder structure.

Across all reviewed samples, feedback for failing criteria was typically framed as actionable guidance: it identified the condition targeted by the check, described how the submission deviated from the expected behavior, and suggested a concrete next debugging step without revealing a complete solution. When rubric reconstruction faithfully captured the intent of the

underlying check, the resulting feedback was usually localized and test-case-specific rather than generic. In this sense, the workflow often translated autograder structure into more interpretable natural-language explanations.

Feedback quality depended strongly on rubric reconstruction fidelity. When rubric items preserved the key intent and constraints of the original checks, feedback aligned well with the autograder's structure, with each rubric item producing a focused explanation tied to a specific check. When rubric items only partially captured a check or merged multiple behaviors, the resulting feedback could become misaligned, either by emphasizing the wrong aspect of the code or by justifying judgments that differed from the original autograder. In a subset of reviewed mismatch cases, these feedback discrepancies also helped surface potential brittleness or underspecification in the original autograder, suggesting that the workflow may have value not only for grading support but also for autograder auditing.

Discussion

One notable pattern in the results is the imbalance between false positives and false negatives: disagreements occurred more often when the autograder failed a criterion but the LLM marked it correct. Several interpretations are plausible. In some cases, the autograder may be overly strict or underspecified, enforcing narrow implementation choices that exclude valid alternatives. In other cases, the reconstructed rubric may partially capture the intent of a check while omitting a key constraint, leading the grader to over-credit a submission relative to the original autograder. A third possibility is genuine LLM judgment error, particularly on edge cases that are explicitly targeted by instructor-authored tests. Distinguishing among these explanations is important because they imply different interventions: revising the autograder, improving rubric reconstruction fidelity, or strengthening the grading prompt and validation workflow.

This pattern also helps explain one of the more interesting qualitative observations from RQ3: disagreement cases can be informative beyond simple grading mismatch. In some reviewed cases, rubric-based feedback exposed potential brittleness or underspecification in the original autograder, suggesting that the workflow may support not only feedback generation but also autograder auditing and refinement. From an instructional perspective, this is important because instructors who already have autograders in place may be able to use this workflow to surface clearer grading criteria, make grading intent more transparent, and provide feedback that is closer to rubric-based explanation without redesigning assessments from scratch.

Threats to validity and limitations

The scope of the study is also limited. Our results are drawn from a single, long-running, asynchronous Python course on Codio, so generalization to other programming languages, course contexts, assignment types, or grading styles remains to be tested. In addition, preprocessing choices, specifically deduplication of repeated observable outcomes and sampling

improved diversity and controlled compute cost, but they may have reduced the representation of rare failure modes or edge cases.

Finally, the workflow was evaluated using a single model family and a structured one-shot prompting setup, and results may vary under different models, prompting strategies, or generation settings. A further limitation is that feedback quality has not yet been systematically quantified at scale.

Future work

Several next steps follow directly from the findings of this study. First, a targeted error analysis is needed to better understand disagreement cases, especially the observed imbalance between false positives and false negatives. One useful direction would be to sample disagreement cases by assessment and rubric criterion and classify them into categories such as autograder strictness, rubric reconstruction mismatch, and LLM judgment error. Stratifying this analysis by autograder type may help identify where alignment is most likely to degrade.

Feedback quality needs to be evaluated more systematically at scale. This could be achieved by sampling across agreement and mismatch categories with rubric-guided human coding, and calibrated automated scoring, to examine how feedback quality changes across different task types and failure modes.

Another particularly promising direction is to combine rubric reconstruction with execution-backed evidence through an MCP-enabled architecture. When instructor tests are available, those tests could be executed and their structured results fed into the feedback pipeline to improve reproducibility and reduce hallucination risk. When tests are absent, candidate tests could be generated from rubric criteria, executed, and used as additional evidence for grading and explanation. An execution-backed approach may improve auditability while preserving the rubric-centric feedback framing explored in this paper.

Conclusion

This paper examined an LLM-enabled grading workflow that reconstructs existing autograder checks into rubric-style criteria and then uses those criteria to evaluate historical student submissions at scale. Across 70 Python assessments, rubric reconstruction produced near-complete mappings between autograder checks and rubric items when autograder artifacts could be successfully exported and parsed. Using these reconstructed rubrics, the LLM grader achieved 96.0% criterion-level agreement with the platform autograder across approximately 41,000 criterion evaluations, indicating that rubric-based LLM grading can align closely with existing automated judgments. These results suggest that existing autograder logic can be repurposed as a foundation for more interpretable, rubric-oriented grading and feedback workflows without requiring instructors to redesign assessments from scratch.

References

[1] U. Alkafaween and I. Albluwi, “Automating autograding: Large language models as test suite generators for introductory programming,” arXiv:2411.09261 [cs.CY], 2024. [Online]. Available: https://arxiv.org/abs/2411.09261

[2] D. Bengtsson and A. Kaliff, “Assessment accuracy of a large language model on programming assignments,” B.S. thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1779792

[3] P. Denny et al., “Computing education in the era of generative AI,” Commun. ACM, vol. 67, no. 2, pp. 56–67, Jan. 2024, doi: 10.1145/3624720.

[4] Z. Fan, S. H. Tan, and A. Roychoudhury, “Concept-based automated grading of CS-1 programming assignments,” in Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA), Seattle, WA, USA, 2023, pp. 199–210, doi: 10.1145/3597926.3598049.

[5] J. Gao, B. Pang, and S. S. Lumetta, “Automated feedback framework for introductory programming courses,” in Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ. (ITiCSE), Arequipa, Peru, 2016, pp. 53–58, doi: 10.1145/2899415.2899440.

[6] G. Haldeman et al., “Providing meaningful feedback for autograding of programming assignments,” in Proc. 49th ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE), Baltimore, MD, USA, 2018, pp. 278–283, doi: 10.1145/3159450.3159502.

[7] Q. Hao and M. Tsikerdekis, “How automated feedback is delivered matters: Formative feedback and knowledge transfer,” in Proc. IEEE Frontiers in Education Conf. (FIE), Covington, KY, USA, 2019, pp. 1–6, doi: 10.1109/FIE43999.2019.9028686.

[8] H. Keuning, J. Jeuring, and B. Heeren, “A systematic literature review of automated feedback generation for programming exercises,” ACM Trans. Comput. Educ., vol. 19, no. 1, Art. no. 3, Sep. 2018, doi: 10.1145/3231711.

[9] J. Leinonen et al., “Comparing code explanations created by students and large language models,” in Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ. (ITiCSE), Turku, Finland, 2023, pp. 124–130, doi: 10.1145/3587102.3588785.

[10] J. Leinonen et al., “Using large language models to enhance programming error messages,” in Proc. 54th ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE), Toronto, ON, Canada, 2023, pp. 563–569, doi: 10.1145/3545945.3569770.

[11] M. Messer, N. C. C. Brown, M. Kölling, and M. Shi, “Automated grading and feedback tools for programming education: A systematic review,” ACM Trans. Comput. Educ., vol. 24, no. 1, Art. no. 10, Feb. 2024, doi: 10.1145/3636515.

[12] F. Nilsson and J. Tuvstedt, “GPT-4 as an automatic grader: The accuracy of grades set by GPT-4 on introductory programming assignments,” B.S. thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1779778

[13] M. Novak and D. Kermek, “Assessment automation of complex student programming assignments,” Educ. Sci., vol. 14, no. 1, 2024, doi: 10.3390/educsci14010054.

[14] J. C. Paiva, J. P. Leal, and Á. Figueira, “Automated assessment in computer science education: A state-of-the-art review,” ACM Trans. Comput. Educ., vol. 22, no. 3, Art. no. 34, Jun. 2022, doi: 10.1145/3513140.

[15] S. Parihar et al., “Automatic grading and feedback using program repair for introductory programming courses,” in Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ. (ITiCSE), Bologna, Italy, 2017, pp. 92–97, doi: 10.1145/3059009.3059026.

[16] A. Pathak et al., “Rubric is all you need: Enhancing LLM-based code evaluation with question-specific rubrics,” arXiv:2503.23989 [cs.SE], 2025. [Online]. Available: https://arxiv.org/abs/2503.23989

[17] T. Phung et al., “Generative AI for programming education: Benchmarking ChatGPT, GPT-4, and human tutors,” in Proc. ACM Conf. Int. Comput. Educ. Res. (ICER), vol. 2, Chicago, IL, USA, 2023, pp. 41–42, doi: 10.1145/3568812.3603476.

[18] J. Prather et al., “The robots are here: Navigating the generative AI revolution in computing education,” in Proc. Working Group Reports Innov. Technol. Comput. Sci. Educ. (ITiCSE-WGR), Turku, Finland, 2023, pp. 108–159, doi: 10.1145/3623762.3633499.

[19] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic generation of programming exercises and code explanations using large language models,” in Proc. ACM Conf. Int. Comput. Educ. Res. (ICER), Lugano, Switzerland (and Virtual Event), 2022, pp. 27–43, doi: 10.1145/3501385.3543957.

[20] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” IEEE Trans. Softw. Eng., vol. 50, no. 1, pp. 85–105, 2024, doi: 10.1109/TSE.2023.3334955.

[21] M. L. Siddiq et al., “Using large language models to generate JUnit tests: An empirical study,” in Proc. Int. Conf. Eval. Assess. Softw. Eng. (EASE), Salerno, Italy, 2024, pp. 313–322, doi: 10.1145/3661167.3661216.

[22] M. Sánchez-Santillán et al., “An empirical evaluation of the formative feedback supported by dashboard in the context of compilation error,” Comput. Appl. Eng. Educ., vol. 31, May 2023, doi: 10.1002/cae.22640.

[23] J. L. Tims, C. Tucker, M. A. Weiss, and S. Zweben, “Computing enrollment and retention: Results from the 2021–22 undergraduate enrollment cohort,” ACM Inroads, vol. 14, no. 4, pp. 24–43, Nov. 2023, doi: 10.1145/3629980.

[24] U.S. Bureau of Labor Statistics, “Computer and information technology occupations,” Occupational Outlook Handbook, 2025. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/[Accessed: Jan. 21, 2026].

[25] J. Wang et al., “Software testing with large language models: Survey, landscape, and vision,” IEEE Trans. Softw. Eng., vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.

[26] K. George, “CERP pulse survey: A snapshot of 2025 undergraduate computing enrollment patterns,” Computing Research News, vol. 37, no. 9, Oct. 2025. [Online]. Available: https://cra.org/crn/2025/10/cerp-pulse-survey-a-snapshot-of-2025-undergraduate-computing-enrollment-patterns/ [Accessed: Jan. 20, 2026].

[27] Otter-Grader, “Otter-Grader documentation,” 2026. [Online]. Available: https://otter-grader.readthedocs.io/en/latest/ [Accessed: Jan. 21, 2026].

[28] D. Blank et al., “nbgrader: A tool for creating and grading assignments in the Jupyter Notebook,” J. Open Source Educ., vol. 2, no. 11, 2019, doi: 10.21105/jose.00032.

[29] G. L. Herman and C. Zilles, “PrairieLearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning,” in Proc. ASEE Annu. Conf. Exposition, Seattle, WA, USA, 2015, doi: 10.18260/p.24575.

[30] A. Frias, S. Krishnakumar, and A. Pandey, “FlexiGrader: An LLM-based personalized autograder to enable flexible and open-ended creative exploration in CS1,” ASEE conference paper, 2025. [Online]. Available: https://ayush-pandey.github.io/documents/DSAI-ASEE2025.pdf [Accessed: Jan. 21, 2026].

[31] Gradescope, “Gradescope autograder documentation,” 2026. [Online]. Available: https://gradescope-autograders.readthedocs.io/en/latest/[Accessed: Jan. 21, 2026].