

Autograder+: A Multi-Faceted AI Framework for Rich Pedagogical Feedback in Programming Education

Vikrant Sahu

Indian Institute of Technology
Bhilai, Chhattisgarh, India
vikrantsahu@iitbhillai.ac.in

Raghav Borikar

Indian Institute of Technology
Bhilai, Chhattisgarh, India
raghavbori@iitbhillai.ac.in

Dr. Gagan Raj Gupta

Indian Institute of Technology
Bhilai, Chhattisgarh, India
gagan@iitbhillai.ac.in

Nitin Mane

Indian Institute of Technology
Bhilai, Chhattisgarh, India
nitingautam@iitbhillai.ac.in

Abstract

The rapid growth of programming education has outpaced traditional assessment tools, leaving faculty with limited means to provide meaningful, scalable feedback. Conventional autograders, while efficient, act as “black-box” systems that merely indicate pass/fail status, offering little instructional value or insight into the student’s approach. **Autograder+**¹, a comprehensive and intelligent framework designed to evolve autograding from a summative evaluation tool into a formative learning platform, addresses this gap through two unique features: (1) **feedback generation** via a fine-tuned Large Language Model (LLM), and (2) **visualization** of all student code submissions. The LLM is adapted via domain-specific fine-tuning on curated student code and expert annotations, ensuring feedback is pedagogically aligned and context-aware. In empirical evaluation across 600 student submissions across various programming problems, Autograder+ produced feedback with an average BERTScore F1 of 0.7658, demonstrating strong semantic alignment with expert-written feedback. To make visualizations meaningful, Autograder+ employs **contrastively learned embeddings** trained on 1,000 annotated submissions, which organize solutions into a performance-aware semantic space resulting in clusters of functionally similar approaches. The framework also integrates a Prompt Pooling mechanism, allowing instructors to dynamically influence the LLM’s feedback style using a curated set of specialized prompts. By combining advanced AI feedback generation, semantic organization, and visualization, Autograder+ reduces evaluation workload while empowering educators to deliver targeted instruction and foster resilient learning outcomes.

¹Code available at: <https://github.com/zvikrnt/Autograder-Plus>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CODS '25, Pune, India

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
<https://doi.org/XXXXXXX.XXXXXXX>

CCS Concepts

• **Applied computing** → **Computer-assisted instruction**; • **Computing methodologies** → *Natural language generation*; • **Human-centered computing** → Information visualization.

Keywords

Programming, Large Language Models, Education, Prompt Engineering, Summarization

ACM Reference Format:

Vikrant Sahu, Dr. Gagan Raj Gupta, Raghav Borikar, and Nitin Mane. 2025. Autograder+: A Multi-Faceted AI Framework for Rich Pedagogical Feedback in Programming Education. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (CODS '25)*. ACM, New York, NY, USA, 10 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

The escalating global demand for computational literacy has triggered a massive surge in enrollment in computer science courses at every level of education. This scaling presents an immense pedagogical challenge [34] providing timely, meaningful, and personalized feedback [48] to an ever-growing body of students. The traditional method of meticulous manual code review by instructors or teaching assistants is logistically untenable in large-scale educational settings.

Automated assessment systems, commonly known as “autograders,” have emerged as a necessary solution to this scalability problem [39]. Platforms such as Gradescope [46] and Autolab automate [32] the execution and validation of student code, enabling immediate, objective, and scalable evaluation of functional correctness. However, this efficiency comes at a significant pedagogical cost. Most automated grading tools rely on dynamic test suites or static analysis and typically provide binary pass/fail outcomes or output comparisons, giving little insight into the student’s approach or conceptual errors. Building comprehensive test suites is time-consuming and incomplete test coverage can produce misleading feedback, failing to diagnose root causes or connect errors to underlying concepts. Consequently, students often engage in trial-and-error loops—making surface-level changes to pass tests without resolving conceptual misunderstandings. These tools, in essence, act as opaque arbiters of correctness, offering minimal educational scaffolding [34]. Beyond these limitations, automatic feedback systems also face challenges of trust: students frequently

doubt the correctness of the system’s evaluations or the reliability of the feedback provided, which can diminish their confidence in the learning process.

Recent work highlights the importance of richer feedback mechanisms. For instance, using context-aware LLMs with structured reasoning [43] approaches—such as chain-of-thought prompting—can provide interpretable evaluations and actionable insights beyond mere correctness [9]. These approaches align more closely with pedagogical [44] goals by diagnosing errors, guiding logic, and assisting conceptual understanding.

This paper contends that a fundamental disconnect exists between the assessment of functional correctness and the cultivation of conceptual understanding. To bridge this divide, we present Autograder+, a comprehensive and intelligent framework designed to evolve autograding from a summative evaluation tool into a formative learning [53] platform. Secure, containerized code execution ensures robustness, while inference latency remains practical, averaging 11–13 seconds per response for selected models. Importantly, before any feedback is delivered to students, the generated output is validated by course instructors or TAs, ensuring that correct and non-hallucinated feedback reaches learners. Instructor-facing analytics further provide **actionable insights** into cohort-level trends, revealing performance patterns across assignments and time. The primary contributions of this work are as follows:

- (1) **A Complete Autograding Pipeline:** An end-to-end, modular framework combining secure sandboxed execution, static/dynamic program analysis, and semantic evaluation for flexible, extensible code assessment.
- (2) **Innovative Feedback Enhancement:** A prompt pooling mechanism that dynamically injects expert-written prompts at inference, improving feedback quality.
- (3) **Concept-Aware Instructor Analytics:** Interactive UMAP visualizations of code embeddings (Fig. 1) learned via contrastive fine-tuning, reveal common strategies, misconceptions, and outliers for targeted pedagogical correction.

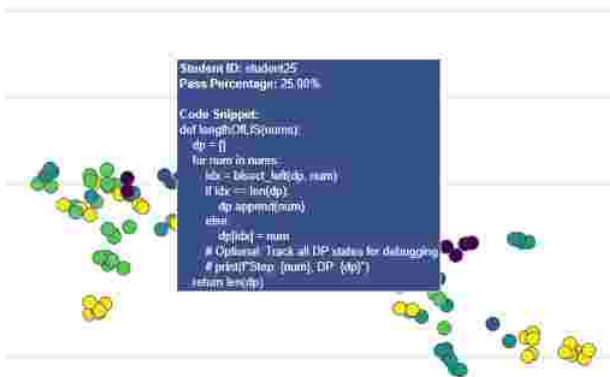


Figure 1: A sample student code submission drawn from Interactive UMAP of code embeddings which are generated via the embedding model

Autograder+ accomplishes this by augmenting traditional program validation with deep semantic analysis, sophisticated AI-driven feedback, and interactive visual analytics for instructors [8]. In empirical evaluation on 600 student submissions, Autograder+ achieved an average BERTScore F1 of 0.75 against TA-written gold-standard feedback, demonstrating that its generated explanations closely align with human instructors.

2 Related Work

The design and philosophy of Autograder+ are built upon a rich history of research spanning automated assessment [23], program analysis, educational data mining, and the revolutionary advancements in artificial intelligence for code [11].

2.1 Traditional Autograding Systems

The foundation of automated assessment in programming was laid by systems prioritizing scalability and objective evaluation[11]. Seminal platforms like Autolab and Gradescope defined the paradigm of test-case-driven assessment, executing student code in a sandbox and comparing outputs against expected results[35]. Their impact has been transformative, enabling instructors to manage assignments in massive open online courses (MOOCs) and large university classes, but the pedagogical model is inherently limited[35]. The “black-box” nature of this testing provides little insight into the student’s cognitive process or algorithmic strategy; feedback typically consists of binary pass/fail signals or cryptic output diffs[35].

2.2 Program Analysis for Educational Feedback

Recognizing the limitations of simple input/output testing, researchers have long sought to “open the black box” using techniques from program analysis. **Static analysis** tools inspect the source code without running it, typically by constructing an Abstract Syntax Tree (AST) [1] or a control-flow graph [31]. This allows for detection of syntax errors, violations of coding style, and structural anti-patterns, enabling systems to provide students with feedback on the form and structure of their code[11]. **Dynamic analysis** tools execute the code to observe its runtime behaviour, catching exceptions, logical errors, and performance issues that static analysis might miss. While these methods offer more granular feedback than traditional autograders, they often focus on technical aspects rather than student intent, and typically fail to diagnose higher-order conceptual errors[35]. Moreover, such feedback is rarely mapped explicitly to curricular learning objectives[2].

2.3 Large Language Models and Prompt Engineering in Education

The advent of large language models (LLMs) pre-trained on vast datasets of code, such as CodeBERT [17], CodeT5+ [52], and the Qwen series [40], has unlocked new possibilities for automated pedagogical support [51]. These models exhibit a profound ability to comprehend, summarize, and generate code [4]. Initial educational applications focused on using these models as “explainers” or “translators”. More recently, a wave of research has explored their potential for generating formative feedback on student programming assignments [38]. These studies confirm that LLMs can

produce fluent, human-like feedback that transcends syntax. However, many of these efforts treat the LLM as a standalone component, disconnected from a robust execution pipeline [30]. Furthermore, controlling the output of these powerful models to be pedagogically sound is a significant challenge. This has given rise to the field of **prompt engineering** [42], where the input given to the model is carefully crafted to steer its output. Our work on Dynamic Prompt Pooling builds directly on this idea, but automates the selection of the steering prompt based on semantic analysis. While others have used LLMs for feedback, Autograder+ distinguishes itself by embedding two distinct, advanced AI modeling strategies within a complete autograding framework, ensuring feedback is grounded in the code's actual runtime behavior and enhanced by dynamic, pedagogically-informed prompt steering [21].

3 AI-Driven Semantic Feedback Models

We now elaborate on the two primary AI model variants that power Autograder+. These models represent distinct strategies for generating high-quality pedagogical feedback. The first is a direct approach focused on fine-tuning a generative model, while the second employs a more sophisticated method of structuring the semantic space through contrastive learning [51].

3.1 Model Variant 1: Fine-Tuned LLM

The first variant represents a direct and powerful approach to feedback generation. It involves taking a pre-trained Large Language Model (LLM), and specializing it through supervised fine-tuning. We leverage a subset of the `nvidia/openreasoningcode` dataset [3] consisting of `problem` & `code` pairs. The model is trained to generate the debugging insight when conditioned on the `student_code`. By learning from examples, the model learns to identify common errors and articulate explanations in a manner that is both conceptually precise and accessible to students. While highly effective, this approach treats the code and its performance as input text, without explicitly structuring the underlying semantic relationships [41] between different student solutions.

3.2 Model Variant 2: Contrastively Fine-Tuned Embedding Model

The second, structurally distinct variant enhances instructor-facing analytics. This approach fine-tunes the embedding model itself to create a performance-aware semantic space where the geometric arrangement of code embeddings reflects their functional correctness. These structured embeddings are then used to drive UMAP visualizations [33], enabling instructors to identify clusters of correct, partially correct, and incorrect solutions, as well as recurring misconceptions or outlier strategies. This is achieved by fine-tuning a base code embedding model using a combination of powerful contrastive loss functions [29].

3.2.1 Multi-Label Supervised Contrastive Finetuning: To create embeddings that are aware of both the problem type (e.g., Fibonacci, Palindrome) and the correctness of the solution (e.g., PASS, PARTIAL, FAIL), we fine-tune a base code embedding model using a **Multi-Label Supervised Contrastive Loss (MulSupCon)**, inspired by the work of Zhang et al. [56]. This approach teaches the model to

group similar solutions in the embedding space based on shared characteristics defined by their labels.

Mathematical Formulation: Let $\mathcal{B} = \{(z_i, y_i)\}_{i=1}^N$ be a batch of N samples, where $z_i \in \mathbb{R}^D$ is the D -dimensional embedding of a code snippet and $y_i \in \{0, 1\}^C$ is its corresponding multi-hot label vector over C classes (e.g., `problem_q6`, `tier_PASS`).

First, we compute the pairwise cosine similarity [28] between all normalized embeddings in the batch. This forms a similarity matrix $S \in \mathbb{R}^{N \times N}$, where $S_{ij} = z_i \cdot z_j$.

These similarities are then scaled by a temperature parameter $\tau > 0$ to produce logits, which control the sharpness of the probability distribution.

$$\text{logits}_{ij} = \frac{S_{ij}}{\tau} = \frac{z_i \cdot z_j}{\tau}$$

The log-probability [47] of correctly identifying a sample j as a positive for an anchor sample i (among all other samples $m \neq i$ in the batch) is given by the log-softmax function:

$$\log P_{ij} = \text{logits}_{ij} - \log \sum_{m=1, m \neq i}^N \exp(\text{logits}_{im})$$

For an anchor sample i and a specific class k , the set of indices of its positive samples, $P(i, k)$, includes all other samples j in the batch that also possess class k .

$$P(i, k) = \{j \in \{1, \dots, N\} \setminus \{i\} \mid y_{ik} = 1 \text{ and } y_{jk} = 1\}$$

The supervised contrastive loss for anchor i with respect to class k^* is the average of the negative log-probabilities over all its positive samples for that class [27]. This loss is only calculated if the positive set $P(i, k)$ is not empty ($|P(i, k)| > 0$).

$$\mathcal{L}_{i,k} = \begin{cases} -\frac{1}{|P(i,k)|} \sum_{j \in P(i,k)} \log P_{ij} & \text{if } |P(i,k)| > 0 \\ 0 & \text{otherwise} \end{cases}$$

The total loss for a single anchor sample i is the sum of its per-class losses, calculated only for the classes it actually possesses. The multi-hot label vector y_i acts as a mask for this summation.

$$\mathcal{L}_i = \sum_{k=1}^C y_{ik} \cdot \mathcal{L}_{i,k}$$

Finally, the total Multi-Label Supervised Contrastive Loss for the entire batch $\mathcal{B}$ is the mean of the individual anchor losses. We only average over anchors that have at least one label to avoid division by zero if a sample has no labels. Let $\mathcal{I}^+ = \{i \mid \sum_{k=1}^C y_{ik} > 0\}$ be the set of indices of anchors with at least one label.

$$\mathcal{L}_{\text{MulSupCon}} = \frac{1}{|\mathcal{I}^+|} \sum_{i \in \mathcal{I}^+} \mathcal{L}_i$$

3.2.2 Multiple Negatives Ranking (MNR) Loss: We further refine the model's ability to distinguish between samples with a complementary training objective by incorporating the **Multiple Negatives Ranking (MNR) Loss** [26], which is highly effective for retrieval-oriented tasks. For a given positive pair (anchor, positive), MNR loss treats all other samples in the batch as hard negatives and uses a standard cross-entropy loss to train the model to assign

a higher similarity score to the positive pair than to any of the negative pairs. This directly optimizes a ranking objective.

We combine these two losses using a weighting factor α :

$$\mathcal{L}_{\text{Total}} = \alpha \cdot \mathcal{L}_{\text{MulSupCon}} + (1 - \alpha) \cdot \mathcal{L}_{\text{MNR}} \quad (1)$$

By training our embedding model with this hybrid loss function, the resulting semantic space becomes highly structured along multiple axes simultaneously. This performance-aware embedding is a far more potent input for the instructor analytics engine.[56]

4 The Autograder+ Framework

Autograder+ is architected as a modular, multi-stage pipeline designed to systematically process student submissions, enriching them at each step with layers of analysis. This design ensures that the final feedback is a holistic synthesis of functional correctness, structural integrity, and deep semantic understanding [51].

4.1 System Architecture

The journey of a student's code through the Autograder+ framework is a structured progression, orchestrated by a series of specialized engines. The core components and data flow are as follows:

- (1) **Code Ingestion:** The process begins with the Ingestor module. It is designed with the flexibility to handle common submission formats. The ingestor reads the source code, links it with the corresponding assignment configuration file—a JSON file specifying the problem description, test cases, execution parameters, and language—and encapsulates this information into a standardized data object that flows through the rest of the pipeline.
- (2) **Static Analysis Engine:** The submission then passes through the Static Analyzer. It performs a fast, low-cost analysis of the code without executing it. By parsing the code into an Abstract Syntax Tree (AST), it can efficiently validate syntax, count key structural elements (e.g., number of loops, function definitions), verify adherence to assignment constraints (e.g., presence of a required function name/absence of forbidden libraries), and flag basic anti-patterns [21]. This stage provides an immediate structural and syntactic health check.
- (3) **Dynamic Execution Engine:** Code that is structurally sound proceeds to the Dynamic Analyzer, the crucible to test functional correctness. To ensure safety, security, and reproducibility, this engine leverages Docker containers. Each test case for a submission is executed in a fresh, isolated container, effectively sandboxing the code to prevent filesystem contamination or network access and to enforce resource limits (CPU, memory). This one-shot container strategy guarantees that each test run is independent and clean. The engine meticulously captures the program's standard output (stdout), standard error (stderr), and exit code, comparing them against the expected outcomes defined in the assignment configuration. This stage delivers the definitive ground truth about the code's runtime behavior [51].

- (4) **The Semantic Core:** Following dynamic analysis, the complete submission package—source code, static analysis results, and the detailed execution trace—is passed to the semantic core. It is here that an LLM is employed to generate feedback. The core contains two key sub-modules:

Embedding Engine: This module uses an embedding model to convert the student's source code into a high-dimensional vector embedding. This embedding captures the code's semantic meaning, abstracting away from surface-level syntax to represent its deeper algorithmic intent. [56].

Feedback Engine: This module orchestrates the generation of pedagogical feedback. It takes the code and the dynamic analysis results. Crucially, it houses the Prompt Pooling mechanism, which enhances the final output. The engine then calls the generative model (Base or Fine Tuned) to produce the final textual feedback. [7].

- (5) **Reporting and Analytics:**

Feedback Generator: This module aggregates all the structured information gathered throughout the pipeline and compiles comprehensive reports. It generates an individual Markdown report for each student, presenting the static analysis, a test-by-test breakdown of dynamic results, and the rich, qualitative AI-generated feedback. It also creates aggregated summaries, such as a class-wide CSV file for grade-keeping and high-level review [25].

Analytics Engine: This final engine serves the instructor. It collects the semantic code embeddings from every submission in the class and uses the Uniform Manifold Approximation and Projection (UMAP) algorithm to project them into an interactive 2D scatter plot. This visualization provides an intuitive map of the class's collective problem-solving approaches, transforming raw submission data into actionable pedagogical insights. [18]

4.2 Feedback Enhancement via Prompt Pooling

A key innovation within the Autograder+ framework is the Prompt Pooling mechanism, which enhances the pedagogical quality of feedback from any underlying generative model. This technique provides a lightweight and dynamic method for steering the LLM's focus at inference time, ensuring that its output is not only technically accurate but also aligned with a specific, contextually relevant instructional goal [10]. The mechanism operates as follows:

- (1) **Curate a Prompt Pool:** A repository of expert-written instructional prompts is created by instructors or curriculum designers. Each prompt is designed to focus an LLM's analysis on a specific programming concept (e.g., recursion base cases, loop termination conditions), error type (e.g., `IndexError`, `TypeError`), or pedagogical strategy [6].
- (2) **Pre-computation of Prompt Embeddings:** At initialization, the framework uses an external embedding generator (e.g., a Sentence-BERT model) to compute a high-dimensional vector embedding for every prompt in the pool. These prompt embeddings are then cached for efficient retrieval [50].
- (3) **Runtime Code Embedding:** When a student submission is processed, the *same* embedding generator is used to create a vector embedding for the student's code snippet.

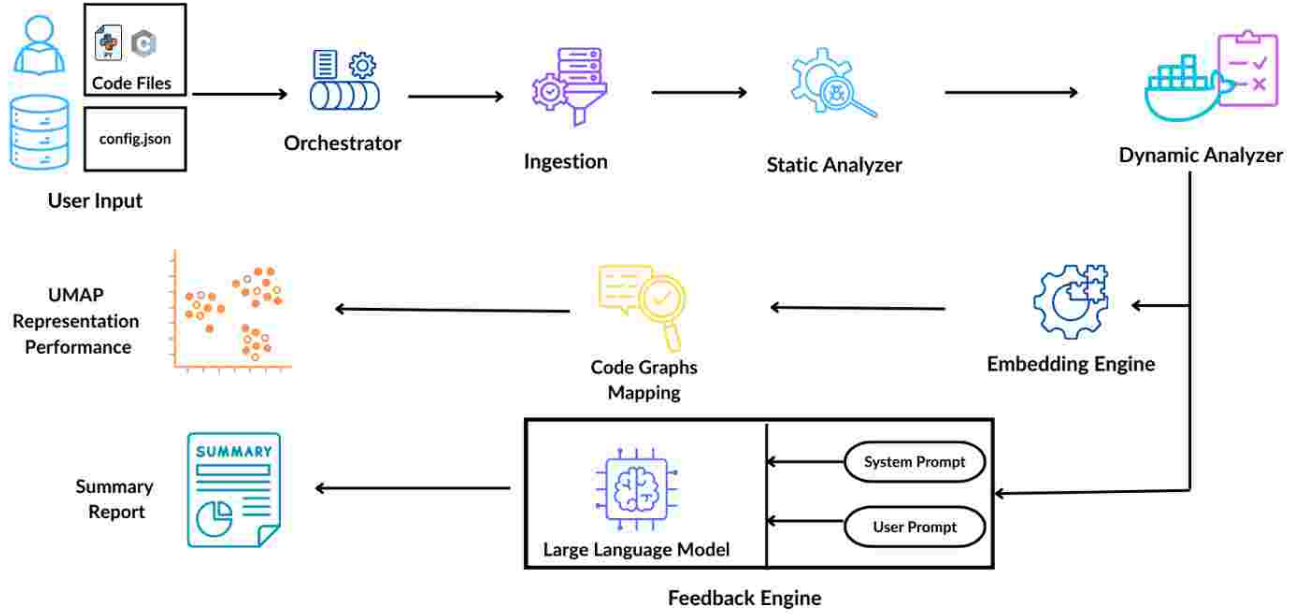


Figure 2: End-to-end architecture of Autograder+.

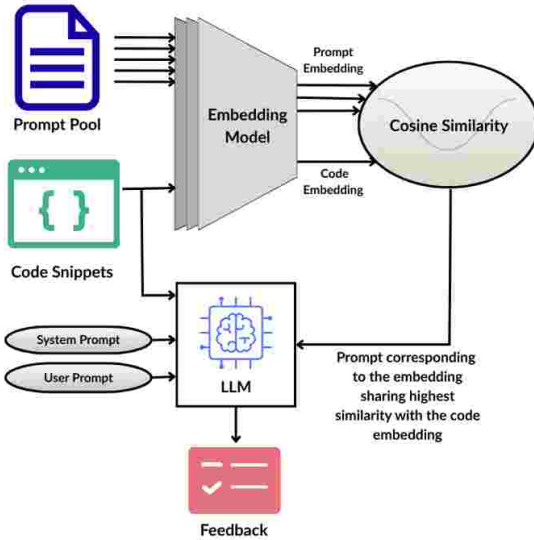


Figure 3: Architecture of the prompt-pooling framework

- (4) **Semantic Similarity Search:** The system then calculates the cosine similarity between the student's code embedding and every cached prompt embedding in the pool. This identifies which instructional prompt is most semantically relevant to the student's specific solution.
- (5) **Dynamic Prompt Injection:** The prompt corresponding to the highest cosine similarity score is selected as the "best-matching" instructional focus. This selected prompt is then appended to the context provided to the generative LLM. The final request sent to the LLM includes the standard system prompt defining the AI's persona [24], the user prompt containing the student's code and error logs (if any), and this dynamically chosen instructional prompt.

This approach provides remarkable flexibility. An instructor can easily refine the system's pedagogical behavior by simply adding or editing the text-based prompts in the pool, allowing the framework to be adapted to different course levels or topics with minimal technical overhead [7].

A representative example of this process is shown in Table 1, where a student's code is analyzed. The framework not only executes the code but also selects a semantically relevant instructional prompt ("loop constructs") and generates feedback that pinpoints the precise logical error. This example demonstrates how Autograder+ moves beyond binary correctness, producing feedback that is both actionable and conceptually targeted.

Table 1: A sample I/O from the framework

Question	Code Snippet	Test Cases Passed	Prompt Selected	Feedback
Implement a function to compute the nth Fibonacci number. The Fibonacci sequence is defined as follows: $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n > 1$. The function should handle both small and large inputs efficiently.	<pre>def fibonacci(n): a, b = 0, 1 for _ in range(n): c = a + b a, b = b, a + b return a</pre>	6/7	Advice: Pay attention to the loop constructs. Check for correct initialization, termination conditions, and potential off-by-one or infinite loop errors.	<output> The issue lies in the line 'a, b = b, a + b', where you're incorrectly updating the values of 'a' and 'b'. It should be 'a, b = c, a + b' to correctly calculate the Fibonacci sequence. </output>
Given an integer array nums, return the length of the longest strictly increasing subsequence. Input: nums = [10,9,2,5,3,7,101,18] Output: 4 Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.	<pre>def lengthOfLIS(nums): if not nums: return 0 dp = [0] * len(nums) for i in range(len(nums)): for j in range(i): if nums[i] > nums[j]: dp[i] = max(dp[i], dp[j] + 1) return max(dp)</pre>	0/4	Advice: Pay attention to dynamic programming concepts. Look for evidence of memoization or tabulation to avoid re-computing results.	<output> It seems like you're comparing elements correctly but the issue might be with how you're handling the DP array. Are you initializing it properly and updating its values accordingly? </output>

5 Experiments and Results

To rigorously evaluate the Autograder+ framework and compare the efficacy of our two primary AI model variants, a comprehensive experimental study was conducted.

5.1 Datasets

To train, validate, and evaluate the proposed AI models, we utilized a combination of internally collected student submissions and a large-scale external code corpus. These datasets serve distinct purposes, as described below.

Table 2: Datasets Collected/Used

Dataset	Source	Size	Purpose
NVidia ORC	External Corpus	171,000 Code Examples	Preliminary Fine Tuning of Base Model
IITBh PW25S	Programming Workshop	1000 Submissions	Contrastive Fine-Tuning of Embedding Model
IITBh PC25	Internal Programming Course	600 Submissions	Evaluation

5.1.1 External Corpus for Foundational Fine-Tuning. To enhance the general code reasoning capabilities of our base generative model we performed fine-tuning using a subset of nvidia/openreasoning code dataset [3], which was further augmented and contained ~ 15,000 code examples.

5.1.2 Institute-Collected Student Submissions. We curated two distinct datasets from our academic and workshop activities, each tailored to a specific modeling approach. [19]

IITBh PW25S: 1,000 student submissions collected during a programming workshop at IIT Bhilai. Each submission was automatically labeled by our Dynamic Analyzer as pass, partial-pass,

or fail based on test-case results. [5]. These labels enabled construction of [anchor, positive, negative] triplets for contrastive training of the embedding model, creating a performance-aware semantic space, where positive samples are drawn from the same correctness class as the anchor, and negative samples are drawn from a different class. This process enables the model to learn a performance-aware semantic space without the need for manual feedback annotation.

IITBh PC25: 600 student submissions collected from internal programming courses. [55] spanning 20 LeetCode-style algorithmic problems (e.g., Fibonacci, Disarium). [15] This dataset is particularly valuable as it contains a diverse range of correct solutions, partially correct attempts, and common logical and syntax errors. [22] Each of them was manually evaluated by Teaching Assistants (TAs), who provided "gold-standard" pedagogical feedback. [14] This collection of (submission, TA_Feedback) pairs served as the primary **validation corpus** for our Feedback models while the expert TA feedback serves as the **reference ground truth** for SBERT and BERTScore based evaluation.

5.2 Evaluation Metrics

The quality of the AI-generated feedback will be quantified using two key metrics that measure semantic alignment with the human-written reference feedback. [13]

SBERT Cosine Similarity: This metric evaluates the semantic similarity at the sentence level. It measures the cosine of the angle between the sentence embeddings of the generated feedback and the reference feedback.

BERTScores: These metrics operate at the token level, computing a similarity score for each token in the generated feedback against tokens in the reference feedback. It provides more granular precision, recall, and F1-scores, capturing lexical overlap in a context-aware manner. [45]

5.3 Main Results

We first evaluated several state-of-the-art large language models (LLMs) integrated directly into the Autograder+ feedback pipeline without any domain-specific fine-tuning or prompt pooling. The aim was to establish a realistic performance baseline against which subsequent enhancements could be measured.[43]

Table 3 summarizes these results. Four key metrics are reported: **BERTScore F1**, **Precision**, **Recall**, and **SBERT Cosine Similarity**—each measuring how closely the AI-generated feedback aligns semantically with gold-standard, TA-written feedback. Higher values indicate closer semantic alignment with expert feedback. [16]

Among the tested models, **falcon3:10b** [49] (0.7435) and **llama3.2:3b** [36] (0.7235) achieve the highest BERTScore F1 values, indicating stronger semantic alignment with human feedback [15], while other models such as qwen3:8b and phi4-reasoning lag significantly in both precision and recall [6]. The SBERT cosine similarity scores, although lower in absolute terms, follow the same trend, with **falcon3:10b** and **llama3.2:3b** leading the chart and inherently providing a contextually relevant feedback even before fine-tuning. [12].

Table 3: Baseline Results: The Results present BERT Scores for the models that were incorporated directly into the framework. (w/o fine tuning or prompt pooling)

Model	Avg. F1	Avg. Precision	Avg. Recall	Avg. Cosine Similarity
qwen3:8b	0.3367	0.3253	0.3492	0.1854
deepseek-coder:33b	0.7212	0.7020	0.7421	0.3241
falcon3:10b	0.7435	0.7485	0.7390	0.3449
llama3.2:3b	<u>0.7235</u>	<u>0.7072</u>	<u>0.7412</u>	0.3667
phi4-reasoning	0.3211	0.3144	0.3289	0.1100

To further refine our selection for subsequent experiments, we evaluated the average inference time of each baseline model when integrated into the Autograder+ pipeline. This metric reflects both the computational cost and the practical feasibility of deploying these models in classroom settings, where real-time or near real-time feedback is crucial.

Table 4 summarizes the inference latency (in seconds per response) observed during baseline evaluation. As expected, reasoning based models such as phi4-reasoning [37] incur significantly higher latency, making them less suitable for scalable deployment while deepseek-coder:33b [20] is moderately efficient and comparatively less aligned with human feedback. Balancing both semantic quality and computational feasibility, we selected **llama3.2:3b** and **falcon3:10b** as candidates for the next stage of our study.

Having established the baseline, we integrated fine-tuning, prompt pooling, and their combination into the Autograder+ framework. Table 5 presents results from these enhanced configurations [5].

Table 5 summarizes the impact of adding question text, prompt pooling, and fine-tuning to our Autograder+ framework. The most consistent improvements come from prompt pooling, which raises both lexical (BERT F1/Precision) and semantic (SBERT cosine) scores across models, with **falcon3-10B (Base)** achieving the best lexical match (F1 = 0.7658, Precision = 0.7706) and **llama3.2-3B**

Table 4: Inference Time Analysis: Average time per response measured in seconds across the baseline models.

Model	Avg. Inference Time (seconds/response)
qwen3:8	63s
deepseek-coder:33b	20.6s
falcon3:10b	<u>13.2s</u>
llama3.2:3b	11.8s
phi4-reasoning	60.3

(**Base**) reaching the highest semantic similarity (SBERT = 0.3924). Including the question text yields smaller, mixed gains, suggesting model sensitivity to input format. Interestingly, fine-tuning did not outperform the strong base models, and in most cases resulted in a slightly reduced performance. We attribute this to the nature of our fine-tuning data, which was augmented and partly synthetic, introducing noise and stylistic artifacts that likely led to overfitting and reduced generalization. Additionally, distributional mismatch between the training prompts and evaluation setup, as well as potential overspecialization during fine-tuning, may have contributed to the observed drops. Overall, prompt pooling emerges as the most robust enhancement, while fine-tuning provides limited benefit under our current (augmented) dataset.

It is also important to note that the comparison involves models of different sizes (3B vs. 10B parameters). While falcon3-10B naturally benefits from its larger capacity, our results show that llama3.2-3B, despite being smaller, can achieve competitive or even superior semantic similarity when combined with prompt pooling. This highlights that architectural choices and configuration strategies can sometimes outweigh raw model scale in the context of automatic grading.

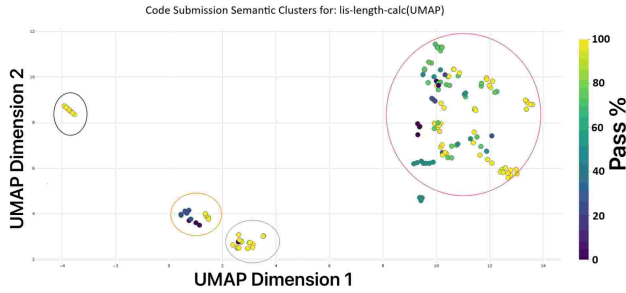
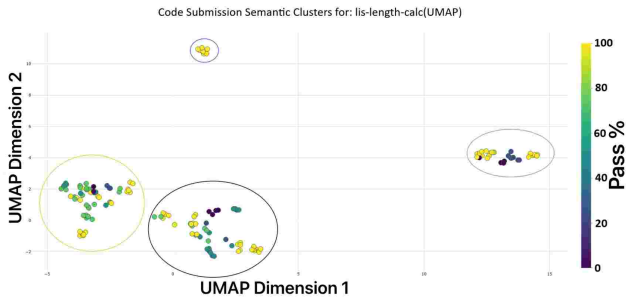
Beyond numerical evaluation, understanding how students approach problems and where misconceptions cluster is critical for targeted instruction [18]. To complement the quantitative metrics in Tables 3 and 5, we leverage the performance-aware embeddings generated by Autograder+ to visualize entire cohorts' solution spaces. Using interactive UMAP projections, code embeddings reveal distinct clusters of correct, partially correct, and incorrect solutions when attempted using diverse approaches, enabling instructors to spot recurring error patterns, strategies used, and isolated outlier cases at a glance. The following section presents these qualitative analytics, illustrating how Autograder+ transforms raw performance data into actionable pedagogical insight [54].

5.4 Qualitative Analysis: Instructor Analytics via UMAP

A key result of our framework is its ability to generate actionable insights for instructors. Using the performance-aware embeddings produced by our contrastively trained embedding model, we generated 2D visualizations of student submissions using the Uniform Manifold Approximation and Projection (UMAP) algorithm. In Fig 5, effect of Contrastive Fine-Tuning is clearly visible as distinct

Table 5: Results across various configurations for Base and Fine Tuned Models

Model	Type	Question	Prompt Pool	Avg. BERT F1	Avg. BERT Precision	Avg. SBERT Cosine
falcon3:10b	Base	X	X	0.7361	0.7372	0.3307
		✓	X	0.7435	0.7485	0.3449
		✓	✓	0.7658	0.7706	0.3725
	FT	X	X	0.7128	0.6812	0.3298
		✓	X	0.7092	0.6714	0.3317
		✓	✓	0.7340	0.7286	0.3459
llama3.2:3b	Base	X	X	0.7134	0.6951	0.3506
		✓	X	0.7235	0.7072	0.3667
		✓	✓	0.7452	0.7315	0.3924
	FT	X	X	0.7056	0.6851	0.3321
		✓	X	0.7182	0.6954	0.3537
		✓	✓	0.7369	0.7321	0.3788

**Figure 4: UMAP projection of embeddings of code as generated by the embedding model before contrastive fine tuning. Each point is a single submission, shaded by its performance (e.g., Light=PASS, Dim=PARTIAL PASS, Dark=FAIL)****Figure 5: UMAP projection of embeddings after contrastive fine tuning with the points shaded by their performance tier (e.g., Light=PASS, Dim=PARTIAL PASS, Dark=FAIL)**

clusters based upon different approaches taken to solve the question have evolved in the UMAP generated via contrastively fine-tuned embedding model. The clusters represent various distinct approaches taken to solve a particular question.

6 Future Work

While Autograder+ demonstrates a novel integration of AI-driven semantic feedback, prompt pooling, and performance-aware visual analytics, several avenues are open for expansion and refinement: **Classroom Deployment:** Implement Autograder+ in programming courses, evaluate its practical impact on learner experience, feedback quality, and instructional workflows. This will provide the evidence of its effectiveness and reveal challenges in integration with existing teaching practices.

Longitudinal Learning Analytics: Assess its effects on problem-solving strategies, misconceptions, and self-efficacy, using temporal UMAPs to track individual and cohort evolution with time.

Large-Scale Evaluation: Deployment across diverse institutions and track its long-term impact on learner performance, scalability, and instructor adoption.

Cross Domain Generalization: Extend adaptability beyond introductory programming to domains like systems, DSA etc.

7 Conclusion

We presented Autograder+, designed to address the core challenges faced by faculty in large-scale programming courses: balancing scalability with meaningful, individualized feedback. By combining domain-specific LLM fine-tuning, performance-aware semantic embeddings, and dynamic prompt pooling, the system produces feedback that diagnoses functional errors and surfaces underlying conceptual gaps in a manner aligned with instructional goals. For instructors, Autograder+ offers actionable, visual analytics that reveal common misconceptions, alternative solution strategies, and at-risk students early in the learning process. This enables targeted interventions, reduces grading overhead, and frees faculty time for deeper engagement with students. Our next steps involve piloting it in our own programming courses to test its feasibility, refine its components, and collect empirical data on its effectiveness. If successful, the framework's modular design allows adaptation to other disciplines where structured problem-solving and conceptual mastery are central, making it a potentially sustainable, faculty-centric solution for delivering high-quality feedback at scale.

References

- [1] Abanoub E. Abdelmalak, Mohamed A. Elsayed, David Abercrombie, and Ilhami Torunoglu. 2025. An AST-guided LLM Approach for SVRF Code Synthesis. arXiv:2507.00352 [cs.SE] <https://arxiv.org/abs/2507.00352>
- [2] Ruben Acuña and Ajay Bansal. 2022. Using Programming Autograder Formative Data to Understand Student Growth. In *2022 IEEE Frontiers in Education Conference (FIE)*. 1–8. doi:10.1109/FIE56618.2022.9962650
- [3] Wasi Uddin Ahmad, Sean Narenthiran, and Somshubra Majumdar. 2025. OpenCodeReasoning: Advancing Data Distillation for Competitive Coding. (2025). arXiv:2504.01943 [cs.CL] <https://arxiv.org/abs/2504.01943>
- [4] Mohammad Akyash, Kimia Zamiri Azar, and Hadi Mardani Kamali. 2025. StepGrade: Grading Programming Assignments with Context-Aware LLMs. arXiv:2503.20851 [cs.SE] <https://arxiv.org/abs/2503.20851>
- [5] Nico Andersen, Julia Mang, Frank Goldhammer, and Fabian Zehner. 2025. Algorithmic Fairness in Automatic Short Answer Scoring. *International Journal of Artificial Intelligence in Education* (2025). doi:10.1007/s40593-025-00495-5
- [6] Anonymous. 2025. From Feedback to Formative Guidance: Leveraging LLMs. In *Proc. ACM ICER*. doi:10.1145/3708319.3733808
- [7] Seyyed Kazem Banihashem, Omid Noroozi, Hassan Khosravi, Christian D. Schunn, and Hendrik Drachsler. 2025. Pedagogical framework for hybrid intelligent feedback. *Innovations in Education and Teaching International* (2025). doi:10.1080/14703297.2025.2499174
- [8] Patrick Bassner, Eduard Frankford, and Stephan Krusche. 2024. Iris: An AI-Driven Virtual Tutor for Computer Science Education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024)*. ACM, 394–400. doi:10.1145/3649217.3653543
- [9] Antonin Berthon and Mihaela van der Schaar. 2025. Language Bottleneck Models: A Framework for Interpretable Knowledge Tracing and Beyond. arXiv:2506.16982 [cs.CL] <https://arxiv.org/abs/2506.16982>
- [10] Sumie Tsz Sum Chan, Noble Po Kan Lo, and Alan Man Him Wong. 2024. Enhancing university level English proficiency with generative AI: Empirical insights into automated feedback and learning outcomes. *Contemporary Educational Technology* 16, 4 (2024). doi:10.30935/cedtech/15607
- [11] Sébastien Combéfis. 2022. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* 1, 1 (2022), 3–30. doi:10.3390/software1010002
- [12] Brendan Cowan, Yutaka Watanobe, and Atsushi Shirafuji. 2024. Enhancing Programming Learning with LLMs: Prompt Engineering and Flipped Interaction. In *Proceedings of the 2023 4th Asia Service Sciences and Software Engineering Conference (Aizu-Wakamatsu City, Japan) (ASSE '23)*. Association for Computing Machinery, New York, NY, USA, 10–16. doi:10.1145/3634814.3634816
- [13] Wei Dai, Yi-Shan Tsai, Jionghao Lin, Ahmad Aldino, Hua Jin, Tongguang Li, Dragan Gašević, and Guanliang Chen. 2024. Assessing the proficiency of large language models in automatic feedback generation: An evaluation study. *Computers and Education: Artificial Intelligence* 7 (2024), 100299. doi:10.1016/j.caeai.2024.100299
- [14] Erkan Er, Gökhan Akçapınar, Alper Bayazit, Omid Noroozi, and Seyyed Kazem Banihashem. 2025. Assessing student perceptions and use of instructor versus AI-generated feedback. *British Journal of Educational Technology* 56, 3 (2025), 1074–1091. doi:10.1111/bjet.13558
- [15] Estévez-Ayres, I., Callejo, P., and Hombrados-Herrera. 2024. Evaluation of LLM Tools for Feedback Generation in a University Programming Course. *International Journal of Artificial Intelligence in Education* (2024). doi:10.1007/s40593-024-00406-0
- [16] D. Federiak. 2024. Prompt Engineering as a New 21st Century Skill. *Frontiers in Education* (2024). doi:10.3389/feduc.2024.1366434
- [17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] <https://arxiv.org/abs/2002.08155>
- [18] Tania Amanda Nkoyo Frederick Enye, Chukwuebuka Fortunate Ijezie, Ahmad Imam Amjad, Maaz Amjad, Sabur Butt, and Gerardo Castañeda-Garza. 2025. Advances in Auto-Grading with Large Language Models: A Cross-Disciplinary Survey. In *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)*. Ekaterina Kochmar, Bashar Alhafni, Marie Bexte, Jill Burstein, Andrea Horbach, Ronja Laarmann-Quante, Anais Tack, Victoria Yaneva, and Zheng Yuan (Eds.). Association for Computational Linguistics, Vienna, Austria, 477–498. doi:10.18653/v1/2025.bea-1.35
- [19] Alekzander D Green. 2025. AN ANALYSIS OF LLM USE IN INTRODUCTORY PROGRAMMING EDUCATION AND DEVELOPMENT OF AI RESISTANT ASSESSMENTS VIA CODE REVIEWS. (6 2025). doi:10.25394/PGS.29189570.v1
- [20] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] <https://arxiv.org/abs/2401.14196>
- [21] Jayant Havare, Varsha Apte, Kaushikraj Maharajan, Nithin Chandra Gupta Samudra, Ganesh Ramakrishnan, Srikanth Tamilselvam, and Sainath Vavilapalli. 2025. Ai-Based Automated Grading of Source Code of Introductory Programming Assignments. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 171–181. doi:10.1109/ICPC66645.2025.00025
- [22] Michael Henderson, Margaret Bearman, Jennifer Chung, Tim Fawns, Simon Buckingham Shum, Kelly E. Matthews, and Jimena de Mello Heredia. 2025. Comparing Generative AI and teacher feedback: student perceptions of usefulness and trustworthiness. *Assessment & Evaluation in Higher Education* (2025). doi:10.1080/02602938.2025.2502582
- [23] Yann Hicke, Anmol Agarwal, Qianou Ma, and Paul Denny. 2023. AI-TA: Towards an Intelligent Question-Answer Teaching Assistant using Open-Source LLMs. arXiv:2311.02775 [cs.LG] <https://arxiv.org/abs/2311.02775>
- [24] Jamiu Adekunle Idowu. 2024. Debiasing Education Algorithms. *International Journal of Artificial Intelligence in Education* 34 (2024), 1510–1540. doi:10.1007/s40593-023-00389-4
- [25] Lucas Jasper Jacobsen and Kira Elena Weber. 2025. The Promises and Pitfalls of Large Language Models as Feedback Providers: A Study of Prompt Engineering and the Quality of AI-Driven Feedback. *AI* 6, 2 (2025). doi:10.3390/ai6020035
- [26] Addison Jadwin and Catherine Huang. 2023. Improving minBERT Performance on Multiple Tasks through In-domain Pretraining, Negatives Ranking Loss Learning, and Hyperparameter Optimization.
- [27] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. *Advances in Neural Information Processing Systems* 33 (2020), 18661–18673.
- [28] Alfirna Rizqi Lahitani, Adhistya Erna Permasari, and Noor Akhmad Setiawan. 2016. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *2016 4th International Conference on Cyber and IT Service Management*. 1–6. doi:10.1109/CITSM.2016.7577578
- [29] Chungpa Lee, Sehee Lim, Kibok Lee, and Jy yong Sohn. 2025. On the Similarities of Embeddings in Contrastive Learning. arXiv:2506.09781 [cs.LG] <https://arxiv.org/abs/2506.09781>
- [30] Jung X. Lee and Yeong-Tae Song. 2024. College Exam Grader using LLM AI models. In *2024 IEEE/ACIS 27th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 282–289. doi:10.1109/SNPD61259.2024.10673924
- [31] Xiaoqi Li, Yingjie Mao, Zexin Lu, Wenkai Li, and Zongwei Li. 2025. SCLA: Automated Smart Contract Summarization via LLMs and Control Flow Prompt. arXiv:2402.04863 [cs.SE] <https://arxiv.org/abs/2402.04863>
- [32] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. arXiv:2203.09095 [cs.SE] <https://arxiv.org/abs/2203.09095>
- [33] Leland McInnes, John Healy, and James Melville. 2020. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. arXiv:1802.03426 [stat.ML] <https://arxiv.org/abs/1802.03426>
- [34] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Transactions on Computing Education* 24, 1 (Feb. 2024), 1–43. doi:10.1145/3636515
- [35] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Transactions on Computing Education* 24, 1 (Feb. 2024), 1–43. doi:10.1145/3636515
- [36] Meta AI. 2024. Llama-3.2-3B Hugging Face Model Card. <https://huggingface.co/meta-llama/Llama-3.2-3B>
- [37] Microsoft. 2025. Phi-4-reasoning Hugging Face Model Card. <https://huggingface.co/microsoft/Phi-4-reasoning>
- [38] Aditya Pathak, Rachit Gandhi, Vaibhav Uttam, Arnav Ramamoorthy, Pratyush Ghosh, Aaryan Raj Jindal, Shreyash Verma, Aditya Mittal, Aashna Ased, Chirag Khatri, Yashwanth Nakka, Devansh, Jagat Sesh Challa, and Dhruv Kumar. 2025. Rubric Is All You Need: Improving LLM-Based Code Evaluation With Question-Specific Rubrics. In *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1 (ICER '25)*. ACM, 181–195. doi:10.1145/3702652.3744220
- [39] Andre Fabiano Pereira and Rafael Ferreira Mello. 2025. A Systematic Literature Review on Large Language Models Applications in Computer Programming Teaching Evaluation Process. *IEEE Access* 13 (2025), 113449–113460. doi:10.1109/ACCESS.2025.3584060
- [40] Qwen, , An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuyang Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical

- Report. arXiv:2412.15115 [cs.CL] <https://arxiv.org/abs/2412.15115>
- [41] Dmitri Roussinov, Serge Sharoff, and Natalya Puchnina. 2023. Fine-tuning language models to recognize semantic relations. *Language Resources & Evaluation* 57, 4 (2023), 1463–1486. doi:10.1007/s10579-023-09677-w
- [42] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2025. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927 [cs.AI] <https://arxiv.org/abs/2402.07927>
- [43] Johannes Schneider, Robin Richner, and Micha Riser. 2023. Towards Trustworthy AutoGrading of Short, Multi-lingual, Multi-type Answers. *International Journal of Artificial Intelligence in Education* 33 (2023), 88–118. doi:10.1007/s40593-022-00289-z
- [44] Niklas Scholz, Manh Hung Nguyen, Adish Singla, and Tomohiro Nagashima. 2025. Partnering with AI: A Pedagogical Feedback System for LLM Integration into Programming Education. arXiv:2507.00406 [cs.CY] <https://arxiv.org/abs/2507.00406>
- [45] Priscylla Silva and Evandro Costa. 2025. Assessing Large Language Models for Automated Feedback Generation in Learning Programming Problem Solving. *arXiv preprint* (2025). <https://arxiv.org/abs/2503.14630>
- [46] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. GradeScope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) (*L@S '17*). Association for Computing Machinery, New York, NY, USA, 81–88. doi:10.1145/3051457.3051466
- [47] Tommaso Soru and Jim Marshall. 2025. Leveraging Log Probabilities in Language Models to Forecast Future Events. arXiv:2501.04880 [cs.CL] <https://arxiv.org/abs/2501.04880>
- [48] Xiaohang Tang, Sam Wong, Marcus Huynh, Zicheng He, Yalong Yang, and Yan Chen. 2024. SPHERE: Scaling Personalized Feedback in Programming Classrooms with Structured Review of LLM Outputs. arXiv:2410.16513 [cs.HC] <https://arxiv.org/abs/2410.16513>
- [49] Falcon-LLM Team. 2024. The Falcon 3 Family of Open Models. <https://huggingface.co/blog/falcon3>
- [50] En-Qi Tseng, Pei-Cing Huang, Chan Hsu, et al. 2025. CodEv: An Automated Grading Framework Leveraging Large Language Models for Consistent and Constructive Feedback. In *arXiv preprint*. <https://arxiv.org/abs/2501.10421>
- [51] En-Qi Tseng, Pei-Cing Huang, Chan Hsu, Peng-Yi Wu, Chan-Tung Ku, and Yihuang Kang. 2024. CodEv: An Automated Grading Framework Leveraging Large Language Models for Consistent and Constructive Feedback. In *2024 IEEE International Conference on Big Data (BigData)*. 5442–5449. doi:10.1109/BigData62323.2024.10825949
- [52] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs.CL] <https://arxiv.org/abs/2305.07922>
- [53] Yuchen Wei, Dennis Pearl, Matthew Beckman, and Rebecca J. Passonneau. 2025. Concept-based Rubrics Improve LLM Formative Assessment and Data Synthesis. arXiv:2504.03877 [cs.LG] <https://arxiv.org/abs/2504.03877>
- [54] Qunai Xu, Yijia Liu, and Xue Li. 2025. Unlocking student potential: How AI-driven personalized feedback shapes goal achievement, self-efficacy, and learning engagement through a self-determination lens. *Learning and Motivation* 91 (2025), 102138. doi:10.1016/j.lmot.2025.102138
- [55] M. Yousef, K. Mohamed, and W. Medhat. 2025. BeGrading: Large Language Models for Enhanced Feedback in Programming Education. *Neural Computing and Applications* 37 (2025), 1027–1040. doi:10.1007/s00521-024-10449-y
- [56] Pingyue Zhang and Mengyue Wu. 2024. Multi-Label Supervised Contrastive Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 16786–16793. doi:10.1609/aaai.v38i15.29619