FlexiGrader: an LLM-based personalized autograder to enable flexible and open-ended creative exploration in CS1

Alex Frias[1], Shrivaikunth Krishnakumar[1], and Ayush Pandey*[1]

[1]University of California Merced

Abstract
Computer Science courses often rely on programming assignments for learning assessment. Automatic grading (autograding) is a common mechanism to provide quick feedback to students and reduce teacher workload, especially in large classes. However, traditional autograders offer limited personalized feedback and often require all students to solve the same predefined problem, restricting creativity. In this paper, we address these limitations by developing an AI-based autograder that (1) can grade diverse, open-ended assignments where students work on independent, creative projects, enabling a new set of assessments in CS1 (introductory programming) courses, and (2) provides personalized feedback using large language models (LLMs). We present the design of a new assessment strategy in introductory programming courses where each student works on an open-ended problem for their summative assessment. We design generalized scaffolds (project proposal, schematic development, pseudocode, integration of files, and graphs) for these open-ended assessments so that each student completes a project of desired complexity. Existing autograders require rigid structure of inputs and outputs, and therefore, cannot grade such assessments. Our tool, FlexiGrader, integrates code execution verification and unit testing tailored to the specifications of each student individually, followed by code analysis using LLMs to generate feedback and grades. FlexiGrader is capable of handling submissions from large classes and ensures flexibility in grading free-form assignments, making it easier for instructors to design and assess varied projects. The input requirements of our tool is a cover sheet that describes the individualized project, provides paths to external files, and describes the inputs needed to run the program that the student submits. Beyond this student-driven cover sheet, FlexiGrader provides options for the instructor to describe the grading rubric and choose the criteria that will be graded by the AI model to allow flexibilty of a hybrid grading approach where it may be desirable for some criteria to be human graded while other could be graded by the AI model. We hypothesize that live implementation of FlexiGrader in CS1 classrooms can enhance student self-efficacy and creativity in CS education by fostering independent project development. We plan to study this hypothesis in future research. Additionally, we discuss the operational costs of our autograding system, its compatibility with existing autograding frameworks, and the current limitations of our approach. By enabling more creative and personalized assignments, FlexiGrader has the potential to transform assessment practices in introductory computer science courses.

*Email: ayushpandey@ucmerced.edu

1 Introduction

It is well established in computer science (CS) education literature [1], that learning-by-doing and rigorous practice are effective for students to gain programming expertise. Consequently, the formative and summative assessments in CS courses often take the form of programming tasks. These assignments usually have fixed inputs and outputs, making them amenable to autograders — software-based automated grading. Autograders provide immediate feedback to students, reduce teachers' workload, and if implemented carefully, can be fair to all students. These properties make autograders effective for fostering a mastery learning framework [2] in CS education. For example, students can keep on working on mastering a skill in an autograded formative loop until they get the desired grade.

We identify two main limitations to prevalent autograders — lack of personalization and inability to grade open-ended assignments. While some autograders allow for the incorporation of personal feedback from the teaching staff, they often do so at the cost of their efficiency. Some of the most recent autograders also provide feedback messages to students on their programs, however, they have not yet been applied widely and their educational outcomes are not yet fully measured. Further, auto-graded assignments and their solutions must follow a pre-defined rigid structure that limits the creative exploration. To enable richer hands-on experiences, instructors may choose to assign open-ended projects. Learning-oriented assessment studies [3] have shown that such open ended assignments are close to students' interests and can lead to an increase in student engagement and agency.

Recent innovations aim to address these limitations by integrating ML and NLP technologies into autograding systems. These advancements enable tools to assess nuanced aspects of code, such as design patterns, code readability, and logical structure [4]. For instance, ML models can analyze code comments and programming styles to provide more personalized and detailed feedback.These systems balance the efficiency of automation with the depth of personalized evaluation, particularly for open-ended and creative assignments [5]. Furthermore, peer grading systems and ML-based similarity detection are being explored to handle diverse outputs in open-ended projects. These innovations hold promise for fostering creativity and higher-order thinking in students, a domain where traditional autograders have fallen short.

In this paper, we describe the development of a new autograder that can grade open-ended assignments. We implemented such assignments for an engineering computing course at our university with 207 students enrolled in the class. This resulted in 190 Python projects where each project had unique goals, inputs, and outputs that were driven by students' personal interests. Feedback from student evaluations was positive and students reported an increase in their enthusiasm for the course material. However, human grading of these open-ended assignments proved to be exceedingly time-consuming, requiring $\sim$ 40 hours of professors' time and $\sim$ 120 TA hours to grade effectively. Although such assignments could imbibe higher-order thinking in students, human grading is not scalable. The development of the autograder tool in this paper is targeted at addressing this grading and assessment gap.

1.1 Related Tools

The growth of the educational technology industry has come up with autograders, which are widely used in CS1/2 settings around the world. Some of the common products are: zyBooks, Perusall, CodeGrade, CodePost, Codio, Crowdmark, and Gradescope. Most of the above-mentioned advanced tools come with modern user interfaces and smoothly integrate with learning management platforms allowing automatic creation of gradebooks. Most of these tools fundamentally use unit-test-based core auto-grading methodology, where students' work is scored by multiple unit tests. Other advanced features that are fast becoming common for these tools include API-based customization, in-class usage of autograder, and grading of handwritten assignments. Most commercial autograder products provide end-to-end assessment solutions with code review, data analytics, automated integrations, and code completion for students.

On the other hand, open-source autograders are available to instructors who require customized solutions for their course needs. For instance, OtterGrader [6] provides an abstraction API for grading Python and R assignments, thus enabling educators to create grading workflows particular to their needs. A number of grading software have also been developed to suit other programming languages such as WebCAT for C++ and Java, PrairieLearn for C++ and Python, OK-py, and nbgrader for Python and Jupyter notebooks. Most existing autograders provide absolute scores; however, questions such as: ”Did the student produce code that adheres to recommended best practices for the programming language?” cannot be evaluated using common unit testing based autograders. Similarly, student feedback is limited to predefined messages from failed unit tests, offering little guidance for improvement. The recently published Pedal framework [7] attempts to address this limitation by using code structure analysis to provide better targeted prebuilt feedback. By identifying common anti-patterns observed in classrooms, pedal can provide more nuanced feedback to students alongside feedback on rubric-based evaluations such as ”Does the code correctly use a for loop? ”. Alternatively, another approach [8], combines program repair techniques with automated grading to directly evaluate the logic and structure of student submissions themselves. These innovations can effectively handle diverse solutions and demonstrate a scalable approach for assignments in which students design unique projects, propose problem statements, and implement customized solutions. However, to the best of our knowledge, none of the existing autograder tools can fully evaluate open-ended assignments. Most available tools are designed for well-defined tasks with known outputs and predetermined code structures.

2 Methodology

2.1 Open-ended CS1 assessment

With the general availability of generative AI, summative assessments in CS1 have to be designed with the understanding that the students can use these tools to complete the exams. Rather than switching to timed in-person or similar “stricter” settings to prevent AI misuse, CS educators must innovate the assessments such that generative AI tools can be used collaboratively. Open-ended projects offer a path forward. Research has shown that projects as the main summative assesssment in engineering and computing courses can have a greater positive impact on student learning and career preparation than conventional exams [9]. With open-ended projects, the idea is to go one step further. If a project problem statement is carefully designed by the instructor and provided to the students, generative AI tools could have a much easier time finishing the project, especially in CS1. Identifying a problem, exploring possible solutions, and defining what a good real-world implementation would look like are the creative elements of any project. Therefore, we designed open-ended project-based summative assessment exams for a CS1 course taught in Fall 2023 in a course with total enrollment of 207 for which the description is shown in Table 1.

Table 1: Open-Ended Project Description

Project Assignments	**Description**
Milestone 1	Set up computer
Milestone 2	Propose project idea (or choose from provided list
Milestone 3	Create the program framework and I/O structure
Milestone 4	Integrate loops and required logic
Milestone 5	External data and personalize
Demonstration 1	Oral exam
Demonstration 2	Class/lab presentation
Demonstration 3	Skill quiz
Final Project Submission	All files and cover sheets

The open-ended projects can be designed to be creative and exploratory in nature. We gave a set of constraints to the students for their Python projects but the project goals, specific objectives, and the technical approach were assigned as tasks for the students. The project constraints were:

All students must work on an independent Python project. This project must demonstrate the following five key elements of Python that you learn in CS 1X. An ideal project that grades 100 will

1. *Control the program flow with branching and loops*
2. *Uses correct data structures for optimal computations*
3. *Uses functions for modular code*
4. *Loads data from files (real-world data is preferable) and writes outputs to files (if needed)*
5. *Documents the flow of logic with comments, docstrings, and user-friendly messages*

Beyond these constraints, the students were given the independence to propose a project and develop it for their exam. Two main challenges with this style of summative assessment are: lack of clarity for the students on "what do I do?", and the design of a fair grading strategy and rubrics. To address the first challenge, we provided a list of project ideas to the students. The students were also encouraged to propose their own project ideas. Scaffolds were designed carefully for the project so that students could make consistent progress and achieve the best results.

2.1.1 Milestone assignments as project scaffolds

A total of 5 milestone assignments were designed as scaffolds for this project-based summative assessment shown in Table 1. The second scaffold was a "project proposal" assignment where the students proposed the project. The other milestone assignments addressed each of the constraint in the exam instructions (adding branching, loops, functions, files, and visualization).

2.1.2 Categories of proposed projects

A total of 190 projects were proposed by the students. The projects can be classified into the following categories: game design (tetris, tic-tac-toe, rock paper scissor, adventure games), computational apps (computing math formula, finance calculators), data analysis (COVID-19 data analysis, stock market analysis), and simulations (bus tracker, diet simulator, class registration system). A full list of all projects is available online on GitHub [10].

2.1.3 Grading and Rubric

The second challenge in this assessment design is the grading. A detailed grading rubric was shared with the students before the exam, and a short summary of the rubric is shown in Table 2. Due to space constraints, the details of each rubric item are not shown. Therefore, we designed open-ended project-based summative assessment exams for a CS1 course taught in Fall 2023 in a course with total enrollment of 207. Note: not all students who were enrolled submitted a project.

Table 2: Rubric for Open-Ended Assignment

Criteria	**Ratings**	**Points**
Execution: Does the code run?	20 if yes, 15 for logical errors, 10 if ∼50% runs, 0 if not.	20
Branching: Does the code use if-else and loops as needed?	15 if criteria satisfied, 10 if only one, 0 if none.	15
Modularity: Does the code use functions for modularity?	10 if modular, 5 if improper, 0 if none.	10
Structures: Does the code use data structures as needed?	5 if appropriate DS, 0 if not.	5
Files/Advanced Features: Does the code use files/advanced features?	10 if criteria satisfied, 5 if attempted, 0 if not.	10
Documentation: Is the code well-documented?	5 if comments present, 5 if attempted, 0 if not.	10
Demo: Did the code demo the proposal?	25 if yes, 20 if one missing, 15 if ∼50%, 10 if one feature only, 0 if none.	25
UI: Effective UI?	10 if yes, 0 if not.	10

2.2 Tool Specifications

Flexigrader is an LLM-based autograder designed to assess open-ended Python programming assignments. Python programming assessment is possible with off-the-shelf LLMs such as GPT. But, there are various concerns about whether it is fair and reproducible for all students, accurate, and provides meaningful feedback. These are even more significant concerns for grading assignments where students independently explore their own ideas. Based on preliminary experiments with LLMs for grading, we formulate the following specifications for any AI-powered autograder.

1. The autograder's behavior must be reproducible such that if points are taken off for a student's mistake under a rubric item, then exactly the same points must be taken off for all

other students who made that same mistake.

2. The AI must not hallucinate and take off points for “made up” scenarios or error patterns that did not appear in the students’ code or are not eligible according to programming language rules.

3. The text feedback must be personalized, meaningful, and direct. Generic and long paragraphs of text as feedback that does not convey much information would tend to dissuade the students from using the autograder’s feedback to improve.

Note that these specifications are valid when we are grading students’ work beyond unit testing frameworks, as required by independent assignments.

2.3 Flexigrader development

To achieve the tool specifications in the design of our autograder, we tested out zero-shot, fewshot and system messages on GPT as well as fine-tuned LLMs such as CodeLlama and GPT. For training, we use the student data from 147 of 190 Python projects where each project has unique goals and outputs. For this training data, we use human-graded scores and feedback as the ground truth. We also compare the performance of the fine-tuned models with the off-the-shelf LLMs. Finally, a crucial piece of our autograder is the integration with a code execution framework that evaluates whether each code runs or not. For the final presentation of the tool, we show integrated results with the execution framework to emulate a modular unit testing-based process and LLMs for more nuanced code assessment.

2.3.1 Code execution framework

We design a code execution framework that runs students’ code and assigns scores, akin to a unit-testing based autograder. The main difference in our execution framework and existing autograders is its ability to grade students’ work on independent problem statements, code inputs, imports, and outputs. To facilitate automated grading in this scenario, we ask students to submit a cover sheet along with their assignment on the LMS. The first section on this cover sheet is on external packages used. Students list out the names of all packages or files that their project depends on, if any. The next section is on inputs. In this section, the students are asked to provide a sample set of inputs needed to run their code, each separated by a new line. To enhance the reliability of our execution framework, we employ an LLM to analyze the imports in the students’ code and automatically generate a requirements.txt file for setting up a Miniconda environment when the student does not provide one.

File Name	**Imports**	**Input 1**	**Input 2**	**Input 3**	**Input 4**
`Student_1.py`	`library1, library2`	Value 1	Value 2	Value 3	Value 4
`Student_2.py`	`library1`	Value 1	Value 2	Value 3	Value 4
`Student_3.py`	`library1`	Value 1	Value 2	Value 3	Value 4

Table 3: General structure of input data for the code execution framework for each student. Each row represents a submitted Python file, its imports, and the inputs required for execution.

Our goal with the code execution framework is to develop a fully-functional and self-sufficient autograder, as well as enhance the user-friendliness of our tool. This framework acts as the main interface for the human grader. The grader needs to provide the following items to run the code execution framework:

- the directory path containing all project folders,
- a CSV file detailing the project proposals (short project descriptions of goals),
- the main Python file name,
- any external Python files,
- the rubric criteria,
- external packages (if any), and
- list of sample inputs (automatically extracted from the cover sheet in LMS)

Description	**Main Python File Name**	**Main Python File Code**	**External Python File Name***	**External Python File Code***	**Criteria Description***	**Criteria Rating***	**Imports**	**Input***
Description of the task or project	Name of the primary Python file	Code within the main file	Name of the external Python file(s)	Code within the external Python file(s)	Description of grading criteria	Rating or score for each criterion	Libraries or dependencies used	Sample inputs provided or inferred

Table 4: Generalized column headers used in the dataset. Columns marked with * indicate fields that support multiple values (e.g., External Python File Name* can have multiple entries such as External Python File Name 1, External Python File Name 2, etc.).

With these, FlexiGrader automatically sets up individual Miniconda environments to import the necessary packages and uses subprocesses to emulate user input, testing and capturing any issues in the code. We query our language model to generate a score and feedback, which are then parsed into a CSV, allowing the grader to easily input grades.

2.3.2 Data pre-processing

Multiple data pre-processing and annotation steps were required to curate the dataset for finetuning. We anonymized student names and identifiers from the submitted files. However, even after an initial round of anonymization, we noticed that students had used their names within the code. So, we manually removed all references to student's identifiers from their code and any other external files that they submitted. Since the training dataset contains students' work and instructors comments/feedback on it, we also had to remove students' names from the feedback. To effectively train the LLM, we had to fill missing gaps in human feedback as not all students received feedback on their work by the human graders. We incorporated feedback by going through each row in our dataset and making sure it correlates to the assignment and adjusting any feedback that is irrelevant or may confuse the model and pick up anti-patterns. In addition, we fleshed out some of the comments so that the model can be descriptive in their responses as well.

2.3.3 Prompt engineering for effective finetuning

Prompt engineering is one of the most important part in designing LLM-based applications. We applied common prompt engineering guidelines [11] published in the literature. Our prompt design is concise and uses emphasized tags $< >$. in order for the model to recognize patterns and relative importance. We used the Alpaca format [12] to create a consistent and structured prompt. For GPT fine-tuning, we used a Chat-based JSON Schema to structure the prompt. An initial challenge with the prompt design was the Non-ASCII characters that students used in their code. Long context prompts also led to discouraging preliminary results. Thus, we ensure that the prompt is straightforward yet descriptive enough to convey our intent and we removed all non-ASCII characters in the manual dataset annotation process. We tested various prompts using the zero-shot LLM and selected the one that provided the desired structure and response before finetuning. A limitation of the prompt is the restriction to the code length since prompts for LLMs have a maximum context length resulting in having 147 python project. One student in the dataset developed a long project that did not fit the maximum prompt size and we had to remove this project's grading from the finetuning. This is an area of improvement for future designs of autograders.

Alpaca Text Structure

```
  "messages": [
     {"role": "system"
     "content": system_content},
     {"role": "user"
     "content": user_content},
     {"role": "assistant"
     "content": assistant_content}
 ]
```

GPT Prompt Structure

```
{
"instruction": combined_content
"input": ""
"output": assistant_content
}
```

Note: `combined_content = system_content + user_content`

system_content	user_content	assistant_content
### Instruction: [Guidelines for grading, specifying criteria and desired feedback.]	### CodeDescription: <description> [Brief description of the project or task.] </description>	### Response: [Feedback for each criterion, including scores and suggestions.] Received Score - [List of scores for each crit-erion.]
	### crtieria_{i} <criteria> [Desc. and eval. points] </criteria>	
	### MainPythonFileCode <code> [Main Python file code here.] </code>	
	### ExternalPythonFile - [filename].py code: <code> [Code from an external file, if applicable.] </code>	

2.3.4 Off the shelf LLM

we conducted experiments using two popular off-the-shelf LLMs: Claude and ChatGPT. We used system messages to enforce a consistent structure across the models, ensuring uniform output formatting that can be easily parsed into the gradebook.

System Message
You are a grading assistant that strictly follows the output format described below:
Response: Criteria_1: <score> out of <total> Feedback: <feedback text>
Criteria_2: <score> out of <total>
Received Score - [<list of scores>]
Do not deviate from this structure. If additional criteria exist, continue the numbering pattern.

Table 5: Example of system message format guideline

2.3.5 LLM finetuning

For finetuning, we chose popular models that excel in analyzing code. Based on literature review, we found that the Llama is appropriate choice for our autograding task as they have versions that specialize in code generation and is also open-source. For instance, CodeLlama Python 34B scored 53.7 on the HumanEval benchmark, demonstrating its strong code understanding.

For the finetuning, we divide the dataset into training-test subsets, as per the best practices of training machine learning models. We use 85% of the student assignments for training, 15% of the assignments were used for test set.

2.3.6 Metric for comparing autograder performance

Our autograding task differs from some of the existing autograders that score 0 or 1 on various categories since our rubric criteria are set at multiples of 5, with some criteria being scored between 0 to 20, while others being scored between 0 to 10 or 0 to 5. In this setting, some of the standard metrics that measure AI model's performance are not directly applicable. So, we define the following metrics to measure the performance of an autograder. These metrics can be generally applied to other autograders that grade with conventional non-binary scoring types.

- Precision: The fraction of scores given by the model that exactly match the human scores i.e. Precision = $\frac{TP}{TP+FP}$ where TP denotes the number of true positives and FP denotes the number of false positives.
- Gross Precision: The fraction of students' who received the same total score by the autograder as by the human grader. GP = T/T_{tot} where T is the number of students who got the same grade and T_{tot} is the total number of students.
- Error exceedance rate: The frequency of autograder errors that exceed the specified delta threshold, Δ. EER = $\frac{\sum_{i=1}^{N} e_i}{N}$, where N is the total number of scores given by the autograder, e_i is an indicator function that equals to 1 if the autograder errors in scoring is more than Δ.
- Recall: The fraction of human scores that the model correctly identifies as positive i.e Recall = $\frac{TP}{TP+FN}$ where TP denotes the number of true positives and FN denotes the number of false negatives.
- Weighted F1-Score: The harmonic mean of Recall and Precision adjusted to account for class imbalances by assigning weights to each class's score according to its proportion of true instances in the dataset. The computation involves two main steps:
  1. Compute the F1-score for each class c

     $$\text{F1}_c = 2 * \frac{\text{Precision}_c * \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}$$

     where Precision_c and Recall_c are the precision and recall of class c.
  2. Compute the weighted sum of all the classes' F1-scores:

     $$\text{F1}_{\text{weighted}} = \sum_{c=1}^{C} \frac{n_c}{N} \cdot \text{F1}_c$$

     where
     - C is the total number of classes
     - n_c is the number of true instances in class c
     - N is total number of instances

  - $F1_c$ is the F1-score for class c

- Mean Absolute Error (MAE): The average of the absolute differences between the predicted and true values, defined as MAE $= \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i|$ where n is the total number of samples, $\hat{y}_i$ is the autograder score, and y_i is the human score .
- BERT Score (CLS-based): Given two texts, a candidate text(model's feedback) x and a reference (human's feedback) $\hat{x}$, we first obtain a single embedding for each text by extracting the CLS token representation from the final layer of a pretrained BERT model(`bert-base-uncased`):

$$\mathbf{z}_x = \mathrm{BERT\_CLS\_Embedding}(x), \quad \mathbf{z}_{\hat{x}} = \mathrm{BERT\_CLS\_Embedding}(\hat{x}),$$

  where each of $\mathbf{z}_x$ and $\mathbf{z}_{\hat{x}}$ is a vector in $\mathbb{R}^d$ (e.g., $d = 768$ for `bert-base-uncased`).
  We then define our "BERT Score" (in this CLS-based approach) as the cosine similarity between these two embeddings:

$$\mathrm{BERT\_Score}(x, \hat{x}) = \frac{\mathbf{z}_x \cdot \mathbf{z}_{\hat{x}}}{\|\mathbf{z}_x\| \, \|\mathbf{z}_{\hat{x}}\|}.$$

  The dot in the numerator denotes the dot product of the two vectors, and $\|\cdot\|$ is the Euclidean norm.

Note that we report these metrics as a percentage score by multipying the fractions by 100.

3 Results

We show the inference results of the finetuned models in Table 6. An important finding is that non-finetuned models outperform the finetuned models with claude-sonnet3.5 being the best.

Table 6: Model Overall scores

Model	Weighted F1-Score (%)	Recall (%)	Precision (%)	Gross Precision (%)	Error Exceedance Rate (%)	MAE	BERT Score (%)	Training Time (sec)	Inference Time (sec)
CodeLlama-34b Finetuned	16.52	19.05	6.96	27.92	72.18	4.32	85.79	1980	1886.02
GPT4o Finetuned	56.07	19.52	19.88	53.25	46.75	2.47	83.72	1031	544740
GPT4o-mini Finetuned	23.92	16.91	15.28	25.97	74.03	4.40	83.85	682	384960
GPT3.5 Fine-tuned	27.96	14.46	13.51	28.57	71.43	4.25	82.5	465	3848760
GPT4o Base	68.86	22.66	22.79	71.43	28.57	1.68	78.95	N/A	544740
GPTo1 Base	67.03	23.84	21.03	70.78	29.22	1.66	75.11	N/A	1.50
GPTo1mini Base	63.73	22.52	16.93	68.71	31.29	1.94	77.40	N/A	2.50
claude-3-5-sonnet Base	71.91	26.14	25.36	73.38	26.62	1.53	80.39	N/A	1.00

Finally we note that CodeLlama-34b's feedback is very human-like as it has the highest BERT Score overall meaning the model's feedback was semantically similar to the human grader. Moreover, CodeLlama excelled in delivering concise, to-the-point feedback, which may be beneficial for quickly identifying key issues in the code. However, GPT models offered more extensive critiques, highlighting a broader range of improvement opportunities and providing detailed explanations of how to enhance the code. This comprehensive feedback is particularly valuable for students, as it not only identifies errors but also serves as a learning tool, guiding them toward better coding practices and deeper conceptual understanding.

Table 7: Feedback comparison

Criteria	Human	GPT-4	CodeLlama	Claude
Branching	(10/15) Doesn't contain for/ while.	(10/15) The code contains branching statements using multiple if conditions to handle different user inputs and program flows. However, it lacks any form of loop (such as for or while) which is required to fulfill the looping aspect of this criterion.	(10/15) There is no while/ for in your script.	(15/15) The code effectively uses both branching statements (if-else) and loops. Multiple if statements are used for input validation and control flow (e.g., checking restaurant choice validity, order completion). While loops are used for the main program loop and order-taking process. The implementation is appropriate and well-structured.
Modularity	(0/10) The entire script is a sequence of instructions executed linearly, without any functions.	(0/10) The submitted code does not contain any function definitions. All the logic is written in the global scope without modular separation. Introducing functions would improve code organization, reusability, and readability…	(5/10) There is no function in your script.	(10/10) The code demonstrates excellent use of functions with four well-defined functions: `display_menu()`, `take_order()`, and `main()`. Each function has a clear purpose and is properly implemented with appropriate parameters and return values.

Criteria	Human	GPT-4	CodeLlama	Claude
Structures	(0/5) A list of list for materials is not optimal. In fact, a dictionary is a much better data structure here.	(5/5) The code appropriately uses lists to store material properties. The materials list of lists is a suitable data structure for managing multiple materials and their associated attributes. This organization allows for efficient lookup and iteration, which is particularly useful when comparing user input to available materials.	(0/5) `int(5)` is not the best way to store data.	(5/5) The code makes appropriate use of data structures. It uses a nested dictionary for the menu (restaurant as the outer key, menu items as inner keys), which is an excellent choice for this data organization. Lists are used appropriately for enumeration, and strings for text display.
Advanced Features	(5/10) Partial points for the encryption algorithm.	(0/10) The code does not incorporate file reading/writing, graphing, multiple modules, or other advanced features. The project solely focuses on console input/output and basic encryption logic without leveraging any advanced Python features. As a result, this criterion is not satisfied.	(5/10) no file use.	(0/10) The code does not utilize file operations or other advanced features. The menu data is hardcoded; this could be improved by reading from a CSV file, as mentioned in the project description.

Criteria	Human	GPT-4	CodeLlama	Claude
Document	(0/5) There is no comment in your script.	(0/5) The code lacks comments or docstrings. There are no inline comments explaining key lines of code, and no docstrings for functions (noting that no functions are defined either). Good documentation is important to explain logic and improve maintainability.	(0/5) There is no comment in your script.	(5/5) The code is well-documented with clear comments. It includes a title comment and function-level comments explaining the purpose of each major component. These are meaningful and aid in understanding the code's structure and logic.
Demo	(15/25) A basic set of features are implemented. The printed history does not make sense because only the results are printed.	(25/25) The code accurately reflects the project description. It implements all key features such as arithmetic operations, trigonometric calculations, history viewing, and saving history to a file. This comprehensive coverage indicates that most of the proposed features are successfully demonstrated.	(15/25) A basic set of features are implemented.	(20/25) The code implements most of the core features, including restaurant selection, menu display, order-taking, and total calculation. However, it doesn't implement the CSV file reading feature mentioned in the project description. Overall, the implemented features work well and offer good functionality.

3.0.1 Tool availability

FlexiGrader is open-source under the MIT license and will be released along with the paper. The dataset used to train the model contains student data and cannot be released publicly according to the research compliance office at our university. However, we are planning to release a blueprint that contains fully de-identified student data so that other instructors can adopt our models for similar assessment implementation in their CS1 courses.

3.1 Discussion

3.1.1 Impact of autograders on student learning

We anticipate that student self-efficacy could be enhanced in courses that use AI-powered autograders such as the one discussed in this paper. An important premise to this assumption is that students are often driven by their agency to learn [13]. Therefore, working on independent assignments for which they get fair grades/feedback could lead to enhanced confidence and self-belief in completing programming tasks beyond the classroom. Another hypothesis worth exploring is the impact on student engagement with long-term vs short-term (the conventional weekly) assignments. Specifically, are the students more engaged in learning the material when they are working on a project that is based out of their interests and/or personal data that they collect as compared with fixed weekly assignments for the class? Studying this question at an MSI is even more crucial as students are often dedicating very limited amount of time to school due to family and work responsibilities. The role of our autograder on providing detailed, personalized, and regular feedback on such open-ended assignments would be crucial to explore. Such feedback would high throughput compared to human grading and might present more creative learning opportunities. A longitudinal study of student reflections and group interviews could provide useful data to study this question.

3.1.2 Conclusion

We present FlexiGrader — an LLM-based grader that can automatically grade flexible, independent, and open-ended assignments. We tested different state of the art code generation and evaluation large language models and studied their performance in grading this unique assessment style. In conclusion, based on our data analysis and software deisgn, we present a tool that is ready to be adopted for other CS1 courses along with a code execution framework that mimics conventional unit-testing based grading but for open-ended assessments.

References

[1] T. J. Feldman and J. D. Zelenski, "The quest for excellence in designing cs1/cs2 assignments," *ACM Sigcse Bulletin*, vol. 28, no. 1, pp. 319–323, 1996.

[2] J. Garner, P. Denny, and A. Luxton-Reilly, "Mastery learning in computer science education," in *Proceedings of the Twenty-First Australasian Computing Education Conference*, 2019, pp. 37–46.

[3] J. DeNero, S. Sridhara, M. Pérez-Quiñones, A. Nayak, and B. Leong, "Beyond autograding: Advances in student feedback platforms," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017, pp. 651–652.

[4] G. Haldeman and J. Hollingsworth, "Providing meaningful feedback for autograding of programming assignments," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ACM, 2018.

[5] F. Nafa, L. Sreeramareddy, S. Mallapuram, and P. Moulema, "Improving educational outcomes: Developing and assessing grading system (prograder) for programming courses," in *Human Interface and the Management of Information*, Springer, 2023.

[6] U. B. D. S. E. Program, *Otter-grader*, Online. [Online]. Available: https://otter-grader.readthedocs.io/en/latest/.

[7] A. C. Bart and L. Gusukuma, "Autograding python code with the pedal framework: Feedback beyond unit tests," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, 2024, pp. 1893–1893.

[8] R. Parihar, K. Sharma, and S. Kumar, “Automatic grading and feedback using program repair for introductory programming courses,” *Journal of Computing Education Research*, vol. 10, no. 2, pp. 45–62, 2017.

[9] M. Cassens and Y. Reimer, “Engaging cs1 students with project based learning,” in *2018 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2018, pp. 1–5.

[10] Anonymous, *Github for open-ended assessments*, 2024. [Online]. Available: https://github.com/XXX.

[11] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in large language models: A comprehensive review,” *arXiv preprint arXiv:2310.14735*, 2023.

[12] R. Taori, I. Gulrajani, T. Zhang, *et al.*, *Stanford alpaca: An instruction-following llama model*, 2023.

[13] K. Cherbow and K. L. McNeill, “Planning for student-driven discussions: A revelatory case of curricular sensemaking for epistemic agency,” *Journal of the Learning Sciences*, vol. 31, no. 3, pp. 408–457, 2022.