Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions

David H. Smith IV
University of Illinois
Urbana, IL, USA
dhsmith2@illinois.edu

Craig Zilles
University of Illinois
Urbana, IL, USA
zilles@illinois.edu

ABSTRACT

Comprehending and conveying the purpose of code is often cited as being a key learning objective within introductory programming courses. To address this objective, "Explain in Plain English" questions, where students are shown a segment of code and asked to provide an abstract description of the code's purpose, have been adopted. However, given EiPE questions require a natural language response, they often require manual grading which is time-consuming for course staff and delays feedback for students. With the advent of large language models (LLMs) capable of generating code, responses to EiPE questions can be used to generate code segments, the correctness of which can then be easily verified using test cases. We refer to this approach as "Code Generation Based Grading" (CGBG) and in this paper we explore its agreement with human graders using EiPE responses from past exams in an introductory programming course taught in Python. Overall, we find that all CGBG approaches achieve moderate agreement with human graders with the primary area of disagreement being its leniency with respect to low-level and line-by-line descriptions of code.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

GPT-4, Large Language Models, EiPE, Auto-grading

ACM Reference Format:
David H. Smith IV and Craig Zilles. 2024. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653582

1 INTRODUCTION

With the emergence of large language models (LLMs) and the popularity they have gained amongst the general public, many educators have raised questions and concerns regarding their influence on the future of educational practice [5, 16, 17, 19, 26]. These concerns have been raised due to the ease with which everything from essays to code can be generated, which in turn draws into question the integrity of the work students submit [5]. Despite these concerns, there is wide and growing excitement, particularly within the computer science education community, for the potential renaissance these tools may bring by enabling new instructional approaches [3, 6, 8, 20].

Regardless of one's beliefs surrounding LLMs, these tools, particularly those models capable of code generation, are undergoing rapid adoption to increase programming efficiency. GitHub Copilot, which is distributed free to students and educators, has been available since 2020 as a plugin for Visual Studio Code. The release of ChatGPT saw the fastest adoption rate of any platform released, reaching more than a million users in just five days. [1] It now seems inevitable that the usage of these tools will become as commonplace for the next generation of computer science students as StackOverflow has been for the last.

Given the ubiquity and ever growing presence of these tools, this draws into question which skills students should develop in order to utilize them proficiently. Finnie-Ansley et al. [8] have indicated that the ability to formulate a prompt that elicits the correct response from an LLM is a skill that may require explicit instruction. The way in which students interact with a version of OpenAI's GPT, be it through Copilot, ChatGPT, or some other service, is by describing the problem statement that fits the solution they are seeking. Thus, the problem of how to teach and evaluate students' formation of successful queries bears some resemblance to "Explain-in-plain-English" (EiPE) problems, where students are asked to describe a segment of code at a high level.

Existing EiPE autograders perform similarly to trained teaching assistants but lack transparency in their grading mechanism and require labeled training data for each question [9]. To address these limitations, we propose using LLM code generation in an auto-grading pipeline. This pipeline generates code from student responses and evaluates its correctness using unit tests in a process which we term "Code Generation Based Grading" (CGBG). CGBG not only provides feedback by displaying the generated code and the results of unit tests, but also streamlines the EiPE question authoring process by eliminating the need for labeling data and training. Towards informing the use of CGBG, we explore the alignment between the CGBG and traditional EiPE grading methodologies. In doing so, we investigate the following research questions:

RQ1 What is the agreement between trained human raters and code generation based grading?

RQ2 What relationships exist between the features of a given question and the agreement on that question?

ITiCSE 2024, July 8–10, 2024, Milan, Italy

ACM ISBN 979-8-4007-0600-4/24/07
https://doi.org/10.1145/3649217.3653582

[1] https://www.statista.com/chart/29174/time-to-one-million-users/

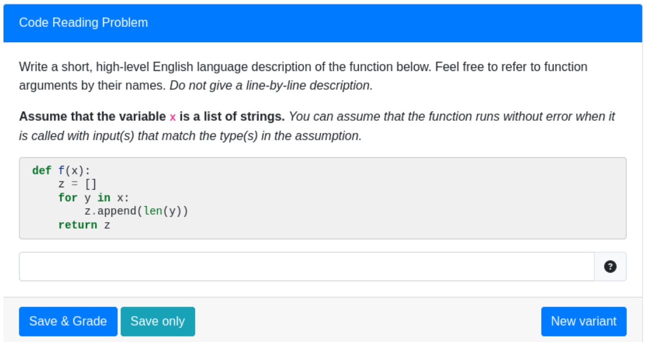


Figure 1: An example of an EiPE question as used in the course from which historical student responses were collected.

2 RELATED WORK

2.1 "Explain in Plain English" Questions

The goal of EiPE questions is to evaluate students' ability to understand and communicate the purpose of a given segment of code [2, 25]. In these questions, students are presented with a segment of existing code and asked to generate a high-level description of what the code does. Prior work has found performance on code comprehension tasks to be highly correlated with other programming skills, such as code writing and tracing [11, 13, 15, 23]. Xie et al. [27] has suggested a sequenced approach in which students are introduced to programming concepts incrementally, the ability to describe code is the penultimate step before code writing. In this way, gaining proficiency in articulating the purpose of code may be placed amongst the pantheon of programming skills introductory courses seek to impart on their students.

However, despite research interest in EiPE questions, they have historically been difficult to scale to larger classrooms, as they typically require manual grading and the development of complicated and subjective rubrics which can be difficult to apply to student responses [2]. The issue of defining an EiPE rubric has been addressed by multiple studies, each attempting to evaluate the ability of students to form an unambiguous, functionally correct, and abstract description of code [4]. In a study investigating faculty perceptions on grading standards for EiPE questions, Fowler et al. [10] found that faculty acknowledged the imprecision of natural language descriptions and placed more value on correctness than instances of slight ambiguity and, in general, preferred high-level descriptions over low-level ones.

Despite the strides that have been made in defining a rubric for EiPE questions, the issue of scale remains. Prior work by Fowler et al. [9] addressed this issue by developing automated grading systems for EiPE questions. Those systems produce results similar to that of a trained human grader. However, this system does suffer in a formative context as it is unable to provide feedback on why a student's response was graded as correct or incorrect. Additionally, it requires a large corpus of human labeled training data that must be constructed on a per-question basis, which adds a layer of difficulty to the process of creating EiPE questions. Given these two shortcomings, this leaves open the door for systems that seek to add both a layer of transparency to the grading process and streamline the process of creating auto-grading mechanisms for EiPE questions.

2.2 Large Language Models and Introductory CS Education

Large language models (LLMs) have already been demonstrated to be a powerful and effective tool for generating code in a variety of contexts, often performing at or above that of the average student on introductory programming problems [6]. A study by Finnie-Ansley et al. [7] evaluated the responses Copilot generated to introductory CS problem, including the (in)famous rainfall problem [24]. The system proved extremely successful, generating a wide variety of solutions to each problem type. Even beyond standard programming problems, LLMs have been shown to be successful, albeit to a lesser extent, at solving Parsons problems [21]. Although many of these studies make note of the existential threat this provides to the current state of CS education (introductory CS education in particular), they also note the avenues that are likely to emerge in the future for leveraging such technologies for teaching.

In a recent study Denny et al. [6] evaluated how various approaches to prompt engineering for CS1 questions impacted GitHub Co-Pilots correctness. In doing so, they found that Copilot successfully generated solutions for approximately 80% of the problems in two attempts. Notably, they found that the two categories of prompts that Copilot had the most difficulty with were those that were both too abstract and too verbose. Within the context of the "Structure of Observed Learning Outcomes" (SOLO) taxonomy and past work evaluating EiPE questions, this may suggest that terse, multi-structural responses are the most effective at eliciting correct responses from Copilot [14].

3 CODE GENERATION BASED GRADING

The "Code Generation Based Grading" (CGBG) process [22] (Figure 2) is divided into three distinct steps. First, a student's response to an EiPE question and a system prompt are combined and fed to GPT-4 to generate a function. That function is then run against a set of unit tests which are manually defined for each question. The students' grades are dependent on the unit tests, receiving a 1 if all tests pass and 0 if any fail. The system prompt format used in this study is as follows:

> Pretend you are an introductory programming student with a rudimentary understanding of programming. You are proficient in the use of functions, loops, conditionals, basic builtin data structures (i.e., list, sets, dictionaries), and user input/output. Construct code as response to the following prompt: Generate a function "studentCode" that *<StudentResponse>*. Respond with the code only and no other explanatory text or example test cases.

Here, the system prompt instructs GPT-4 to pretend to be an introductory student with the intention of it generating code that would be interpretable by students and could therefore serve as feedback to the students with each submission. The system also includes instructions to create a function named "studentCode" to ensure a standardized function name for testing. The student's EiPE response is used to replace `<StudentResponse>`.

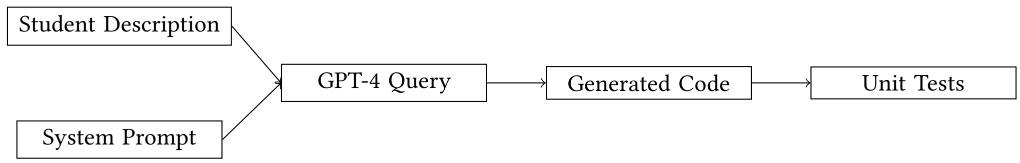


Figure 2: The process of generating code from a prompt and grading it.

This pipeline simplifies the question authoring process for automatically graded EiPE questions to that of traditional code writing questions. Previously developed EiPE auto-graders require the creation of human labeled datasets and training of NLP classification models [1]. This approach also offers the benefit of allowing the instructor to concretely define what aspects of the code and edge cases should be described in the prompt through the unit tests. For example, consider the following code:

```
def foo(x, y):
  for i in x:
    if i == y:
      return y
  return -1
```

If the instructor's goal is to ensure that an explanation for the following code includes an exact description of the function's return behaviour, that can be verified via a test case.

With this process defined, there is one final consideration: how best to accommodate the typically non-deterministic nature of GPT-4. Non-deterministic, in this context, means that given the same prompt GPT-4 may generate a different response each time it is queried. Through the model's temperature parameter, the user can control the "creativity" of the model, with a temperature of 0.0 being deterministic and 1.0 being the most creative. What we wish to avoid is a student submitting a response, which would generally generate correct code, being penalized due to GPT-4 generating an overly creative interpretation of their prompt. To evaluate the best way to prevent such erroneous grading, we put forth and compare the following three methods of grading:

- **Correct at 0.0 Temperature:** Temperature of GPT-4 is set to 0.0 such that the response for a given prompt is deterministic. This single response is then used for grading.
- **Best of 5 at Temperature 0.5:** For a given prompt, GPT-4 is queried 5 times at a temperature of 0.5. If at least one of these responses passes all test cases the student is awarded full marks; otherwise they receive a 0.
- **Majority Vote at Temperature 0.5:** GPT-4 is queried 5 times at a temperature of 0.5. If at least 3 of the 5 responses pass all test cases students are awarded full marks; otherwise, they receive a 0.

4 METHODS

To determine the agreement between human raters and test cases run on the code generated by GPT-4, we begin with exam data collected from an introductory programming course. EiPE questions have been used routinely in the course's proctored exams and have historically been manually graded by the course staff. Typically, teaching assistants (TAs) for the course are trained on grading EiPE responses by longstanding members of the course staff with significant experience with grading EiPE questions. This training process involves grading some answers and meeting to discuss the outcomes. The EiPE responses were typically graded by at least two TAs, with disagreements reconciled to decrease the likelihood of incorrect grading results. These reconciled results constitute the ground truth according the course standards and are what we use to compare to the results generated by CGBG. The course staff grades these question in accordance with a rubric informed by the literature covered in Section 2. This rubric is composed of the following three criteria:

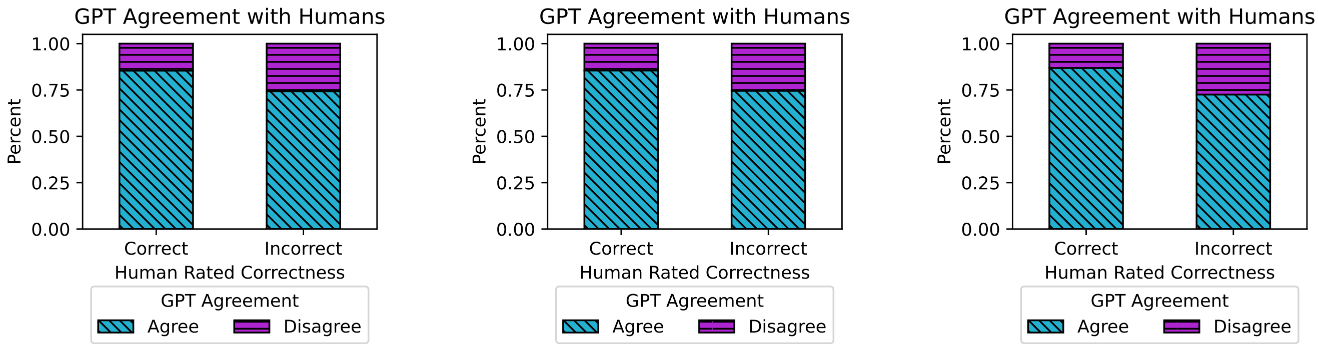
Kappa Value	Agreement
< 0	Less than chance agreement
$0.01 - 0.20$	Slight agreement
$0.21 - 0.40$	Fair agreement
$0.41 - 0.60$	Moderate agreement
$0.61 - 0.80$	Substantial agreement
$0.81 - 0.99$	Almost perfect agreement
1.00	Perfect agreement

Table 1: Standard Cutoffs for Interpreting Cohen's κ

- **Correctness:** The student must describe the process fully such that the process they are describing is functionally correct. If a student is describing a function that filters odd numbers and they incorrectly describe it as filtering even numbers, the response is graded as incorrect.
- **Unabmiguous:** A student must describe the process fully such that multiple, contradictory interpretations of the code is not possible. For example, if a student describes a function as filtering numbers, but does not describe the criteria for filtering, this description would be considered ambiguous.
- **High-Level:** The goal for these EiPE questions is for students to describe the general purpose of code snippets at a high level of abstraction, not provide a line-by-line description. For example, when describing a function that determines if a number is prime, a student should simply state that the function determines if a number is prime, rather than describing the process of iterating over every number less than the input and checking if it is divisible by the input.

These questions are graded on a binary scale where a correct answer receives full marks for meeting all three of these criteria. In total, 6380 student responses from 42 EiPE questions were included in the analysis.

We use Cohen's κ to measure the agreement between the human raters and each of the proposed GPT grading approaches. Cohen's



(a) Single Response at Temp=0 **(b) Majority Vote of 5 Responses at Temp=0.5** **(c) Best of 5 Responses at Temp=0.5**

Figure 3: The agreement between the three approaches to grading with CGBG and human grading. Overall, all appear to have nearly identical levels of agreement and disagreement on responses deemed correct and incorrect by human graders.

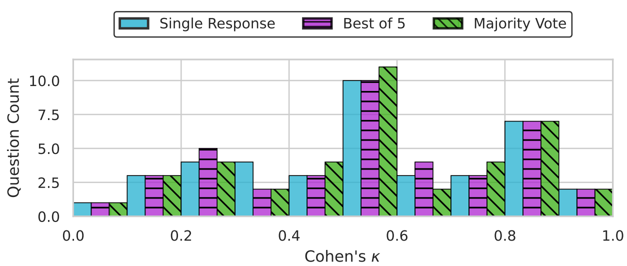


Figure 4: Cohen's κ was computed for each question within each of the three grading methods. Results appear similar for each method with the majority achieving moderate agreement.

$\varkappa$ is a measure of commonly used to evaluate interrater reliability. Interpreting the $\varkappa$ statistic is traditionally done using the scale shown in Table 1 [18]. In our case, we use the same scale to interpret the agreement between the human graders and the results of the test cases run on the code generated by GPT-4.

5 RESULTS

The overall agreement between human graders and CGBG across all questions was similar for each of the three CGBG methods. Grading a single response from GPT-4 at temperature 0 and grading based on majority vote of five responses generated at a temperature of 0.5 both achieved $\kappa = .58$. Grading based on the best of five responses at a temperature of 0.5 was slightly lower at $\kappa = .57$. When looking at where this disagreement occurs for each of the three CGBG methods it appears the majority occurs when CGBG grades a response as correct when human graders graded it as incorrect (Figure 3). This indicates that CGBG has a reasonable agreement with human graders, but, when disagreement occurs, it is more likely to be lenient than strict.

Agreement between human graders and each of the three CGBG methods was computed for each of the questions. The majority of the questions achieved a moderate agreement and all CGBG methods achieved similar results (Figure 4). However, it is notable that there appear to be three distinct "bumps" in the distribution, containing questions that achieved low ($\kappa \leq 0.4$), moderate ($0.6 \geq \kappa > 0.40$), and high ($\kappa > 0.6$) agreement. To gain more insight into areas of agreement and disagreement between CGBG and human graders we next look at the responses to quesitons falling into each of the three categories. Given the similarity of the results between the three grading methods, we will use the results of "correct at 0.0 temperature" for this portion of the analysis.

5.1 Questions with Low Agreement ($\kappa \leq 0.4$)

"Simple" Questions: Most notable among those questions with low agreement are those where the code students were asked to describe was simple. Such questions occur early in the course prior to the introduction of loops and more complex conditional structures. As such, these questions typically involve a function definition and a single line of code that can be described tersely without illustrating the "high-level" purpose of the code. For example, one question asked students to describe the following function which simply returns the average of a list of numbers.

```
def foo(lst):
  return sum(lst)/len(lst)
```

In accordance with the rubric used in the course, descriptions of the above code would be required to be high-level descriptions in order to be marked as correct. For example, a student response that simply states the function returns "*the sum of the element in the list divided by the length of the list*" would be marked as incorrect by human graders as it does not describe the function as "*finding the average*". However, GPT-4 will generate functionally correct code and thus would mark this response as correct. Other questions that suffered from this issue included:

- An absolute value function.
- Returning the maximum of two numbers.
- Determining if x is a multiple of y.
- Returning the range (difference between max and min) of a list of numbers.

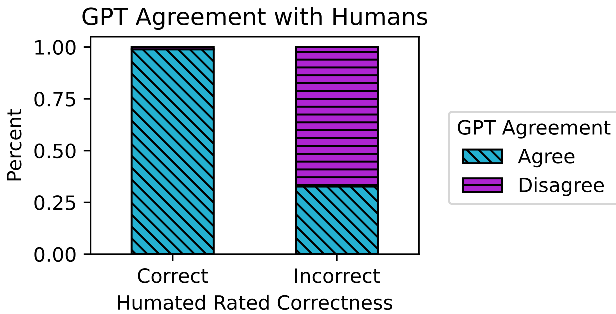


Figure 5: Agreement between human graders and CGBG on questions with "simple" code. CGBG has near perfect agreement with human graders on responses they graded as correct but is more lenient on responses they graded as incorrect, likely due to students correctly describing code but at a low level.

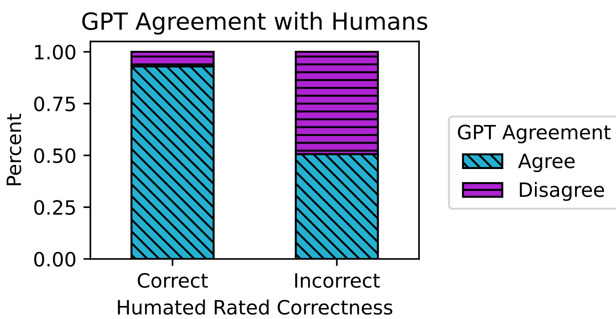


Figure 6: Questions with simple *but recognizable* structure that achieved moderate agreement between CGBG and human graders. We one again see the majority of the disagreement occurs when CGBG is being more lenient than human graders on what a correct response is, likely due to students describing code correctly but at a low level.

Overall, the code generation based grading method was much more lenient than human graders on the aforementioned questions (Figure 5). Looking at the agreement between human graders and CGBG on these questions in the takeaways from this analysis will differ depending on the goal of the rubric being used. If an instructor finds functionally correct but high-level responses to these early questions to be acceptable, then the issue of this grading method being too lenient may not be considered a problem. However, if applying a strict requirement that even simple questions must be answered in a high-level manner, then it would be insufficient to rely on CGBG alone in such cases. Other models or heuristics may need to be placed along CGBG to evaluate difference facets of "correctness".

5.2 Questions with Moderate Agreement ($0.6 \geq \kappa > 0.40$)

Simple but Recognizable Patterns: Similar to the prior section, questions with moderate agreement also struggled with responses

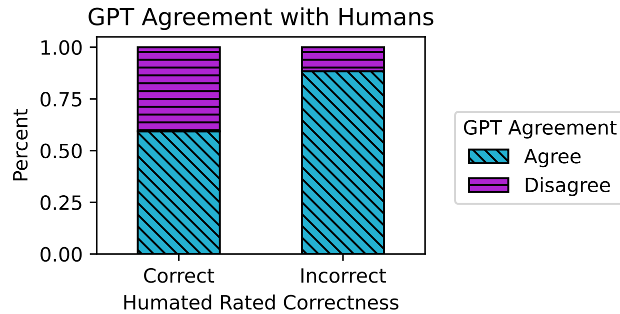


Figure 7: Agreement for questions where CGBG had a large disagreement with human raters on responses they had graded as correct.

that were functionally correct but not sufficiently high-level being graded as correct by CGBG (Figure 6). For example, consider the following code.

```
def foo(x):
  return x % 2 == 0
```

Responses describing this code included:

- "*Returns true if the remainder of dividing 2 from x equals 1*"
- "*True if modulo of x by 2 is 1 false otherwise*"

The human graders mark responses such as these incorrect as they do not describe the higher level purpose of determining if a number is even. Other questions that suffered from this issue included a function that determines if a number is odd and another that determines if a list is empty.

What distinguishes these questions from those covered in the former section is these code snippets cover patterns that students routinely encountered in the course. As such, the superior $\varkappa$ may not be driven by the GPT model being more likely to distinguish between high-level and low-level responses. Rather, these patterns are so common that students are likely to recognize them and use high-level language in their responses, reducing the overall number of low-level responses for these questions.

Context Issues: The only category of questions in which CGBG graded a substantial proportion of incorrect responses that human graders judged to be correct were those where students refer to variables from the code snippet in their responses on questions involving string manipulation (Figure 7). Specifically, a small number of questions that involved slicing ordered collections or iterating between ranges commonly ran into issues with referring to variables by name. For example, consider the following function:

```
def foo(x, y, z):
  return x[x.index(y)+1: x.index(z)]
```

A response of the form "*return the substring that is in between y and z*" would be marked as correct by human graders. Given that GPT-4 is generating code based solely on the student's prompt there exists some ambiguity in what y and z refer to. As such, GPT often generates code that assumes y and z are letters in the string rather than variables and slices between them. For example, GPT-4 generated the following code from such a response,

```
def foo(str):
  return str[str.index("y")+1:str.index("z")]
```

It appears that this issue is specific to questions involving strings as when students refer to variables in problem involving math (e.g., check if x is even) GPT-4 correctly determines that x must be a variable containing a number. With that said, this issue could be mitigated by either providing more context to the GPT-4 model in the system prompt or instructing students not to refer to variables in their responses.

5.3 Questions with High Agreement ($\kappa > 0.6$)

It appears that questions with high agreement lack clusters of traits that can be used to explain the agreement. Instead, the general trend is that these questions have sufficiently complex code that they can not be explained in a line-by-line fashion. For example, the following questions all had $\varkappa$ values above 0.8:

```
def foo(x, y):
    for z in x:
        if z == y:
            return True
    return False
```

```
def foo(x):
    total = 0
    for i in x:
        if i > 0:
            total += i
    return total
```

```
def foo(x):
    tmp = x[0]
    x[0] = x[-1]
    x[-1] = tmp
```

In addition to being sufficiently complex, problems that achieved high agreement generally followed programming patterns that students are explicitly taught in the course. As such, students may be more likely to recognize each of these problems as belonging to those patterns and use correct and sufficiently high-level language to describe them. This would satisfy the rubric used by human graders and be sufficient to generate the correct code using GPT-4.

6 DISCUSSION

Though overall CGBG achieved moderate agreement with human graders, for those who may be interested in implementing CGBG into their classroom or platform it is useful to look more closely at where that disagreement occurred. This is particularly important as understanding what the grading mechanism is evaluating is necessary to inform how it should be utilized. Many of the questions which had the lowest agreement appeared to be questions that occurred early on in the semester and consisted of short functions, typically containing only a single line of code. In these instances, human graders would only grade responses as correct if the code was described at a high level whereas CGBG is capable of generating code whether the response is high or low level. As such, this grading approach could readily be adopted by instructors who are primarily concerned with evaluating the correctness of a description over the degree to which it is high-level, particularly in formative settings.

It may also be the case that CGBG can be integrated into courses alongside traditional EiPE to cover different needs. Existing EiPE auto-grading mechanisms have historically suffered from two core issues: 1) the time investment to label data and train models for autograding questions and 2) transparent grading measures that provide feedback can can be used by the student to deduce why their response was incorrect [12]. CGBG addresses each of these, which is particularly impactful in a formative context where it provides instructors with the ability to quickly author a large, diverse set of problems which provide immediate feedback. CGBG provides a chance to scaffold students by providing the generated code alongside test case feedback and provide insight into how their explanations of code are being interpreted. In turn, this feedback may help improve their ability to explain code, improving their performance on other EiPE related tasks in the course. Future work should explore the impact of this feedback on students' ability to improve their skills on code comprehension tasks.

7 LIMITATIONS

The primary limitation of this work is that the data set of responses from students comes from a course where students are taught to respond to EiPE questions in accordance with the rubric specified by the course staff. Students routinely receive these questions on exams, quizzes, and homework and are therefore very familiar with the grading standards surrounding these questions. This in turn makes it less likely that our analysis would be able to pick up cases where the auto-grader is grading undesirable responses as correct or vice versa.

8 CONCLUSION

Overall, our analysis indicates that there is a reasonable degree of agreement between trained human graders and CGBG. It appears that much of the disagreement comes from CGBG being more lenient than human graders, particularly when asking students to describe code that consists only of a few lines or operations. The one instance where GPT-4 deviated from this trend is where it struggled when students' responses that referenced variables from the code they were describing. This led to ambiguities between variable names and literal values (e.g., variable `e` vs literal "e"). With that said, as much of the disagreement comes from GPT-4 being lenient with descriptions that are correct but somewhat low-level, some instructors with less strict rubrics may find this to be a perfectly reasonable grading mechanism. Future work should consider its utility particularly in formative settings, where students can leverage immediate feedback and multiple attempts to improve both their code comprehension and prompting skills.

ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. DUE 21-21424.

REFERENCES

[1] Sushmita Azad. 2020. *Lessons learnt developing and deploying grading mechanisms for EiPE code-reading questions in CS1 classes.* Ph. D. Dissertation.

[2] Sushmita Azad, Binglin Chen, Maxwell Fowler, Matthew West, and Craig Zilles. 2020. Strategies for deploying unreliable AI graders in high-transparency high-stakes exams. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part I 21*. Springer, 16–28.

[3] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.

[4] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A validated scoring rubric for explain-in-plain-english questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 563–569.

[5] Debby RE Cotton, Peter A Cotton, and J Reuben Shipway. 2023. Chatting and cheating: Ensuring academic integrity in the era of ChatGPT. *Innovations in Education and Teaching International* (2023), 1–12.

[6] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2022. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. *arXiv preprint arXiv:2210.15157* (2022).

[7] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*. 10–19.

[8] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference*. 97–104.

[9] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding" Explain in Plain English" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1163–1169.

[10] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proceedings of the 17th ACM conference on international computing education research*. 69–80.

[11] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* 32, 3 (2022), 355–383.

[12] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. 2021. Attitudes surrounding an imperfect AI autograder. In *Proceedings of the 2021 CHI conference on human factors in computing systems*. 1–15.

[13] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin* 41, 3 (2009), 161–165.

[14] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin* 38, 3 (2006), 118–122.

[15] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[16] Kamil Malinka, Martin Peresíni, Anton Firc, Ondrej Hujnák, and Filip Janus. 2023. On the educational impact of ChatGPT: Is Artificial Intelligence ready to obtain a university degree?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 47–53.

[17] Sathiamoorthy Manoharan, Ulrich Speidel, Anthony Edward Ward, and Xinfeng Ye. 2023. Contract Cheating–Dead or Reborn?. In *2023 32nd Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE)*. IEEE, 1–5.

[18] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[19] Travis Ryan Pickell and Brian R Doak. 2023. Five Ideas for How Professors Can Deal with GPT-3... For Now. (2023).

[20] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. " It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *arXiv preprint arXiv:2304.02491* (2023).

[21] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 299–305.

[22] David H Smith IV and Craig Zilles. 2024. Evaluating Large Language Model Code Generation as an Autograding Mechanism for" Explain in Plain English" Questions. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. 1824–1825.

[23] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*. 117–128.

[24] Michel Wermelinger. 2023. Using GitHub Copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 172–178.

[25] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, PK Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. (2006).

[26] Yunkai Xiao, Soumyadeep Chatterjee, and Edward Gehringer. 2022. A new era of plagiarism the danger of cheating using AI. In *2022 20th International Conference on Information Technology Based Higher Education and Training (ITHET)*. IEEE, 1–6.

[27] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.