



Counting the Trees in the Forest: Evaluating Prompt Segmentation for Classifying Code Comprehension Level

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

Paul Denny
University of Auckland
New Zealand
p.denny@auckland.ac.nz

Max Fowler
University of Illinois
Urbana, USA
mfowler5@illinois.edu

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

Abstract

Reading and understanding code are fundamental skills for novice programmers, and especially important with the growing prevalence of AI-generated code and the need to evaluate its accuracy and reliability. “Explain in Plain English” questions are a widely used approach for assessing code comprehension, but providing automated feedback, particularly on comprehension levels, is a challenging task. This paper introduces a novel method for automatically assessing the comprehension level of responses to “Explain in Plain English” questions. Central to this is the ability to distinguish between two response types: multi-structural, where students describe the code line-by-line, and relational, where they explain the code’s overall purpose. Using a Large Language Model (LLM) to segment both the student’s description and the code, we aim to determine whether the student describes each line individually (many segments) or the code as a whole (fewer segments). We evaluate this approach’s effectiveness by comparing segmentation results with human classifications, achieving substantial agreement. We conclude with how this approach, which we release as an open source Python package, could be used as a formative feedback mechanism.

CCS Concepts

• Social and professional topics → Computing education.

Keywords

Large Language Model, Code Comprehension, EiPE

ACM Reference Format:

David H. Smith IV, Max Fowler, Paul Denny, and Craig Zilles. 2025. Counting the Trees in the Forest: Evaluating Prompt Segmentation for Classifying Code Comprehension Level. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2025)*, June 27–July 2, 2025, Nijmegen, Netherlands. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3724363.3729045>



This work is licensed under a Creative Commons Attribution 4.0 International License. ITiCSE 2025, Nijmegen, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1567-9/2025/06
<https://doi.org/10.1145/3724363.3729045>

1 Introduction

The ability to comprehend and articulate the purpose of code—though always considered a vital skill for novice programmers [18, 28]—has gained new significance in the context of Human-GenAI collaborative coding. A classic activity for assessing code comprehension is the Explain-in-Plain-English (EiPE) question, where students articulate the purpose of a given code snippet—typically a single function—in natural language. While these questions are effective in assessing comprehension [18, 25], they are time-intensive to grade manually [12] and require a high degree of authoring overhead to label data for training auto-grading models [11].

To reduce the authoring overhead of these autograders, Smith and Zilles [23] proposed a code generation-based grading approach. In this method, a student’s response is used to generate code via an LLM, and a suite of unit tests is applied to verify whether the code generated from the response is functionally equivalent to the original code that the student was asked to explain. While this approach effectively distinguishes functionally correct responses from incorrect ones, it is limited in that it does not differentiate between *high-level* responses, which describe the code’s purpose, and *low-level* responses, which focus on its implementation details [9, 22]. This distinction is critical for assessing code comprehension.

In the spirit of the code generation based grading approach—namely through enabling auto-grading approaches that require minimal authoring burden—we propose a novel approach to automatically assess *comprehension level* for responses to EiPE questions. Here, we leverage an LLM to take a student’s response to an EiPE question, segment the response, and map the segments of that response to the lines of code that they describe. The intuition behind this approach being that, if a student provides a high-level description it should map to the code as a whole. However, if they provide a low-level description, this should result in multiple segments where each maps to a different line or group of lines in the code. To evaluate this approach, we address the following research questions:

RQ1: How does classification as *high* and *low* level using the proposed approach compare to human labeling?

RQ2: What post-processing steps can be applied to improve the performance of the segmentation approach?

We release this approach as a feature in the `eiplgrader`¹ package, an open-source Python package for autograding EiPE questions.

¹<https://github.com/CoffeePoweredComputers/eiplgrader>

2 Background

The related work for this study is organized into three areas. Section 2.1 covers foundational programming skills like tracing, explaining, and writing code, focusing on Explain in Plain English (EiPE) questions as tools for assessing code comprehension. Section 2.2 examines the challenges of grading EiPE responses and summarizes autograding methods, noting their strengths and limitations. Section 2.3 provides an overview of natural language programming and prompting, highlighting efforts to integrate prompt engineering into computing education and the gaps our approach addresses.

2.1 Comprehension as a Foundational Skill

Learning to program requires the development of a range of inter-related skills, including tracing, explaining, and writing code [17, 18]. Tracing provides an important foundation, enabling students to understand program flow by simulating execution step-by-step, and explaining builds on this by requiring students to abstract and articulate the purpose of code, moving from a line-by-line analysis to a broader conceptual understanding. Research indicates that tracing and explaining code often precede and support the development of code-writing abilities, highlighting their importance in the learning process [13, 25].

One popular approach for assessing code reading and understanding is the “Explain in Plain English” (EiPE) question type. First described by Whalley et al. [27], EiPE questions were used to assess the comprehension skills of novice programmers, using the SOLO taxonomy as a framework for categorizing and analyzing the natural language responses. Unlike tracing, which focuses on predicting the behavior of individual lines, EiPE tasks require students to articulate the purpose of code segments in natural language, emphasizing abstract reasoning [19].

Despite their pedagogical value, grading EiPE responses has traditionally been time-intensive and subjective. In fact, a survey study by Fowler et al. [12] found that consistent and scalable scoring is one of the primary barriers to the adoption of EiPE questions in programming education, with many educators citing challenges related to the variability in student responses and the effort required to ensure grading fairness. This has led to interest in more scalable approaches for assessing and providing feedback on EiPE tasks [3, 11].

2.2 Grading Explain-in-Plain-English

Although the SOLO taxonomy is often used for classifying answers to EiPE questions, it is often difficult to apply given the need to assess not just abstraction, but also correctness and ambiguity [2]. Achieving consistent and scalable scoring is a well-documented challenge given the subjectivity of grading natural language responses [12]. To address this, several automated grading approaches have been developed. One notable solution involves a logistic classifier trained on human-labeled data to evaluate EiPE responses [11]. While this method achieves grading accuracy comparable to that of human teaching assistants, it requires a substantial upfront investment in labeled datasets for each question. Additionally, given the dichotomous nature of the grading mechanism, it is unable to

provide actionable feedback to students on where their response had shortcomings.

More recently, Smith and Zilles proposed Code Generation Based Grading (CGBG) which leverages large language models (LLMs) to generate code from a student’s natural language response [23]. The generated code is then validated using a suite of instructor-defined unit tests to ensure it is functionally equivalent to the code being explained by the student. This approach removes the need for extensive pre-labeled datasets and provides novel kinds of feedback, including displaying the generated code and test results to students [15, 22]. Additionally, recent evaluations have highlighted the potential for using CGBG in multiple natural languages, thus providing support for non-native English speaking student cohorts [20, 24]. This has led the authors to suggest that, by using this grading approach, the question type is transformed from an “Explain in Plain English” question to an “Explain in Plain Language” question.

However, while CGBG provides objective feedback by executing the generated code, it does not differentiate between surface-level (multi-structural) and deeper (relational) responses [9, 23]. For example, a multi-structural response might describe the behavior of individual parts of the code without identifying their collective purpose, yet would be graded by CGBG as equally correct compared to a relational response that does describe the abstract goal. This distinction is important for identifying students’ comprehension levels and guiding them towards higher-order reasoning. The inability of current autograding systems to assess these cognitive distinctions limits their effectiveness [9].

2.3 Natural Language Programming

Writing natural language explanations of code, as a way to demonstrate code comprehension, is closely related to the idea of natural language programming. Natural language programming involves specifying solutions to computational tasks using natural language—a concept that has been controversial. Dijkstra famously criticized the idea, arguing that the ambiguity of natural language makes it poorly-suited for precise computational tasks [10]. Despite these concerns, recent advancements in large language models (LLMs) have reignited interest in this area, though the translation of abstract ideas into concrete code remains challenging [29].

One critical skill for effective use of LLMs to generate code is the ability to craft clear and precise prompts [6]. Many students find this process difficult, and yet it is emerging as an important new skill [8]. Recent work has explored integrating prompting activities into introductory programming courses, to provide opportunities for students to practice constructing prompts and evaluating the resulting code [15]. One particular approach, “Prompt Problems”, focuses on helping students write prompts that generate functionally correct code [7, 21]. Similar to CGBG, Prompt Problems rely on LLMs to generate code from student prompts, using test cases to evaluate correctness.

Prior work on teaching prompting has primarily focused on functional correctness, with little consideration for aspects of code quality such as style, readability, or efficiency. The work we describe here builds on these efforts, aiming to extend the scope of evaluation beyond correctness to include feedback on the level of comprehension.

System

Task: Create a one-to-one mapping between each segment of a given explanation and the group of lines in the given code which that phrase is associated with. Not all of the description needs to be used. Not all of the code needs to be used. It is very important to only use the words in the user's provided explanation. One segment can map to multiple lines. Here is the code:

```
int sumOfPositives(int arr[], int size) {
    int x = 0;
    for (int i = 0; i < size; i++) {
        if (arr[i] > 0) {
            x += arr[i];
        }
    }
    return x;
}
```

(a) System prompt providing the instructions for the segmentation task and the code to be used.

User

Explanation: input is values with array and length. initially set x to zero, and use for loop to set start i from zero and smaller than length, increasing by 1 for i each run. If values are bigger than zero, then x plus equal values. it will return to x at end.

Assistant

```
{
  "groups": [
    {
      "code": "int sumOfPositives(int arr[], int size) {",
      "explanation_portion": "input is values with array and length"
    },
    {
      "code": "int x = 0;",
      "explanation_portion": "initially set x to zero"
    },
    {
      "code": "for (int i = 0; i < size; i++) {",
      "explanation_portion": "use for loop to set start i from zero and smaller than length, increasing by 1 for i each run"
    },
    {
      "code": "if (arr[i] > 0) {",
      "explanation_portion": "values are bigger than zero"
    },
    {
      "code": "x += arr[i];",
      "explanation_portion": "x plus equal values"
    },
    {
      "code": "return x;",
      "explanation_portion": "it will return to x at end"
    }
  ]
}
```

(b) Explanation and Response Pair 1 – Multi-structural

User

Explanation: sums all positive numbers in the array.

Assistant

```
{
  "groups": [
    {
      "code": "int sumOfPositives(int arr[], int size) {"
      "    int x = 0;"
      "    for (int i = 0; i < size; i++) {"
      "        if (arr[i] > 0) {"
      "            x += arr[i];"
      "        }"
      "    }"
      "    return x;"
      "}"
      "explanation_portion": "sums all positive numbers in the array"
    }
  ]
}
```

(c) Explanation and Response Pair 2 – Relational

Figure 1: The Components of the Prompt Used for the Segmentation Approach Used in this Study.

3 Segmentation Approach

Our segmentation approach—which is done on a per-question basis—is designed to analyze a student's response and map it to the specific line(s) of the code that it describes. This process begins by providing the large language model (LLM) with a *system prompt* (Figure 1a). The system prompt contains three key components: 1) a description of the task it is performing, 2) some guidelines on how it is to perform the task, and 3) the code snippet that the student is to describe.

To further tune the model's performance, we provide the model with two, few-shot examples: 1) an example of a mapping for a multi-structural response (Figure 1b) and 2) an example of a mapping for a relational response (Figure 1c). These examples are used to

help the model understand the difference between the two types of responses and how they should be segmented and mapped to the code. To guarantee adherence to the desired segmentation format in JSON, we leverage OpenAI's GPT-4o with structured output capabilities. This feature allows the model to produce well-formed JSON output that meets the specified requirements².

With the system prompt and several few-shot examples in place, the model is then provided with a student's response to the given question which it segments and maps to the code snippet. The model outputs the resulting JSON which is then used for the classification of the response as either multi-structural or relational.

²<https://openai.com/index/introducing-structured-outputs-in-the-api/>

Write a short, high-level, **English language** description of the following function. Do not give a line-by-line description of the code.

```
int foo(int values[], int length){
    int x = 0;
    for(int i = 0; i < length; i++) {
        if(values[i] > 0) {
            x += values[i];
        }
    }
    return x;
}
```

Assumptions: The input **values** is an array of integer values, and the input parameter **length** is the number of elements in the array.

Note: The code generated by the AI model, based on your description, does not need to be identical to the code presented above. It simply needs to be functionally equivalent, but it may use a different approach.

Create a function foo that...

Provide a high-level description of the code here (250 character limit)

Figure 2: Question interface for A-Q4: Sum of Positive values.

Table 1: Questions for Lab A (Loops, Arrays and Functions) and Lab B (Strings, Text Processing and 2D arrays)

Lab	Question	Description
A	Q1	Sum between a and b inclusive
A	Q2	Count even numbers in array
A	Q3	Index of last zero (see Figure 2)
A	Q4	Sum positive values
B	Q1	Reverse a string
B	Q2	Calculate sum of row in 2D array
B	Q3	Is a vowel contained in a string?
B	Q4	Does a string contain a substring?

4 Methods

The data for this study was sourced from a large introductory programming course at a major research university in Australasia. This course, mandatory for all students in the Faculty of Engineering, is taken during their first year of study and covers standard CS1 topics. At the time of data collection, 889 students were enrolled, with 841 participating. Access to this data was approved by the university’s Human Participants Ethics Committee under protocol number UAHPEC25279.

4.1 Exercises

Students completed a total of 8 exercises across two labs. Each lab contained four exercises. The exercises were designed to cover a range of topics and difficulty levels. The exercises for Lab A (Loops, Arrays and Functions) and Lab B (Strings, Text Processing and 2D arrays) are shown in Table 1. To administer these questions we used the PrairieLearn online-assessment platform [26]. These activities were given in a formative environment where students were given the opportunity to submit multiple times with a maximum of 20 attempts for each question. The questions were autograded using the approach introduced by Smith and Zilles [23]. Students were given feedback on their responses after each submission in the form of the code generated from their response and the results of test cases run on that code.

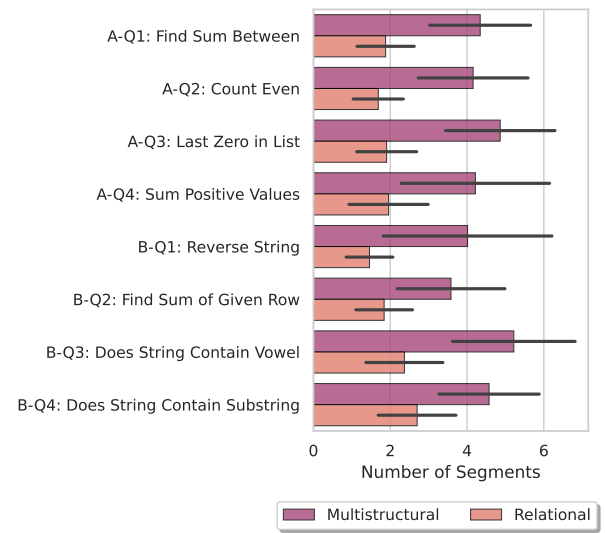


Figure 3: The mean and stdev in the number of segments generated for multistructural and relational responses.

4.2 Human Labeling

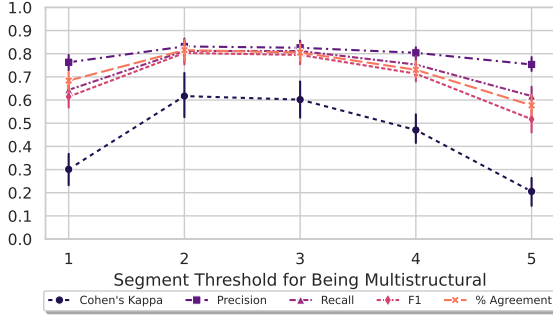
To establish a data set for evaluating the segmentation-based classification of responses, two researchers—using the modified Structure of the Observed Learning Outcome (SOLO) taxonomy [1] introduced by Clear et al. [4]—independently classified a random subset of 200 responses for each question. Beginning with questions A-Q1 and B-Q4, the researchers classified the responses as either *multi-structural correct*, *relational correct*, or *incorrect*. The responses to both questions were categorized and achieved a Cohen’s κ [5] of 0.80 and 0.83, respectively, indicating a substantial agreement [16]. Following reconciliation, the remaining responses were classified individually with each of the researchers classifying half of the responses for each of the remaining questions.

5 Results

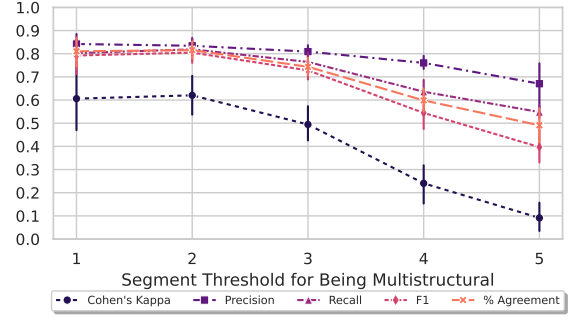
To evaluate differences in segmentation outcomes, we compared the number of segments generated for responses labeled as *multi-structural* versus *relational*. Figure 3 shows that *multi-structural* responses consistently had more segments, aligning with our expectations and the model’s fine-tuning examples. However, it is worth noting that the standard deviations for the segmentation counts is reasonably high, meaning there is a significant overlap between the distributions, which places an upper limit on the degree to which said distributions can be separated. However, the overall results suggest that the segmentation process captures some of the distinctions between *relational* and *multi-structural* responses.

5.1 Performance of Segmentation Classification

Using the resulting segmentation groups, we investigated how varying segmentation thresholds affected the performance of the segmentation-based classification approach. The thresholds ranged



(a) Performance of the segmentation approach for classifying student responses.



(b) Performance of the segmentation after function definition removal post-processing.

Figure 4: Performance of the segmentation approach for classifying student responses. Responses containing a number of segments above each of given thresholds is classified as multistructural and those at or below are classified as relational.

from 1 to 5, where each threshold represents the number of segments above which a response is classified as multi-structural. To evaluate the approach, we calculated Cohen's κ , F1 score, precision, recall, and percent agreement for each threshold.

Initial Segmentation. Figure 4a illustrates the performance of the segmentation approach in classifying student responses. The results indicate that a segmentation threshold of 1 yielded a surprisingly low performance across all metrics, with 68% agreement, a Cohen's κ of 0.3, an F1 score of 0.61, precision of 0.76, and recall of 0.64. The model achieved its highest performance at a threshold of 2, with 81% agreement, a Cohen's κ of 0.61, an F1 score of 0.80, precision of 0.83, and recall of 0.81.

These findings are somewhat counterintuitive, as we expect the presence of fewer segments to be associated with relational responses. Upon closer examination, we found that the model was segmenting students' descriptions of function definitions as separate segments from the rest of the code, leading to a higher number of segments for responses that might otherwise be considered relational. This behavior is inconsistent with the few-shot examples provided to the model, where the entire relational response—including the description of the function definition—is mapped to the entire function, encompassing both its definition and body. In contrast, for multi-structural responses, the description of the function definition (e.g., "defines a function foo that takes...") is mapped only to the line defining the function (e.g., `def foo():`).

Post-processing Improvements. To address this, we implemented a post-processing step that removed segments containing *only* function definitions. The results of this post-processing step, shown in Figure 4b, align more closely with expectations. For a threshold of 1, the percent agreement increased from 68% to 81%. Looking at the results for other metrics, the Cohen's κ improved from 0.30 to 0.61, the F1 score increased from 0.61 to 0.79, precision rose from 0.76 to 0.84, and recall improved from 0.61 to 0.80. The performance for both a threshold of 1 and 2 were roughly equivalent.

6 Discussion

Given the reasonable performance of the segmentation approach, we are left to consider the application of the approach in the context of autograding and feedback. In Section 6.1, we reflect on our findings regarding the improvements seen by the inclusion of the post-processing step of removing segments that only map to the function definitions. Following this, we discuss in Section 6.2 the ways in which this mechanism could be integrated into instruction as a feedback mechanism for students. In both cases, we conclude each section with a summary of the design affordances of our system that enable these features.

6.1 Post-Processing Considerations

Our findings with regard to the post-processing step of removing segments that only map to the function definition highlights the importance of considering the context in which the exercises are used when tuning the autograder. In the course where these data were collected, students were instructed to provide a prompt to the LLM with details on the function definition to avoid ambiguity. This highlights how post-processing steps—be they across all questions or specific to certain questions—can be used to further refine the use of segmentation for classifying responses as multi-structural or relational. For example, consider the following code.

```
int returnIndexOfFirstZero(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

A relational response might be "defines a function that takes an array and its size, returns the index of the first zero if one exists, otherwise, it returns -1." Under the segmentation approach demonstrated in this paper, this would likely be divided into three segments: 1) the function definition, 2) the general behavior, and 3) the default behavior. However, through post-processing, the function definition

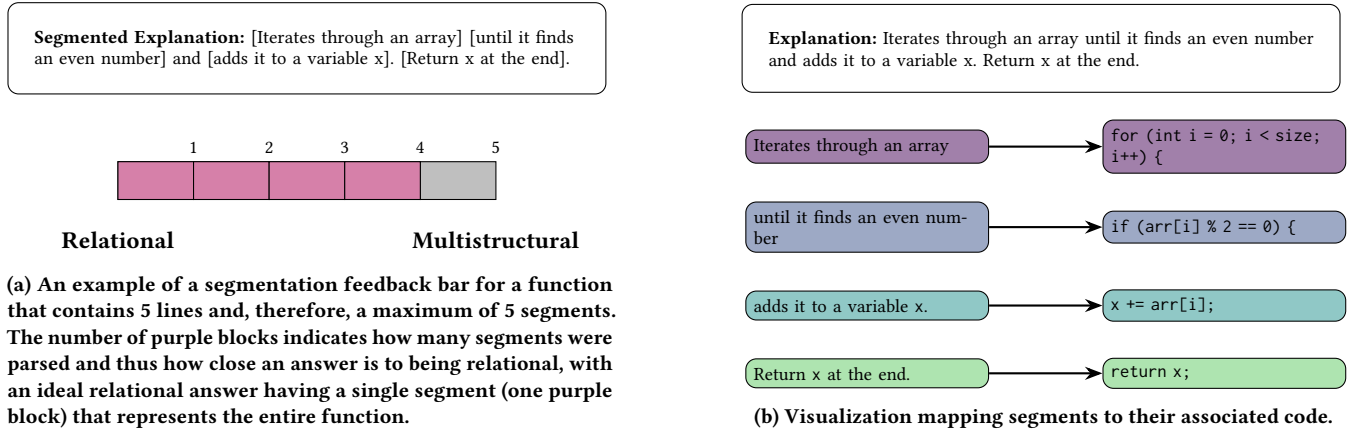


Figure 5: Potential student-facing feedback mechanisms for use with prompt segmentation classification

and default behavior could be removed or accounted for through some other means so the student is not penalized for providing a complete description.

Design Affordances: Our system is designed to offer flexibility through two key features:

- Enabling question authors to provide few-shot examples that align their segmentation expectations.
- Allowing easy post-processing of the structured output to adapt it to the specific context in which the problem is being presented.

6.2 Segmentation as a Feedback Mechanism

We offer two proposed use cases of how this system could be used in a student facing fashion.

The first is simply as a mechanism for providing students with an indication of how far they are from providing a high-level description of code (Figure 5a). As seen from our results, particularly after some degree of post-processing has occurred, guiding students towards responses that contain 1-2 segments is likely to coincide with guiding them away from multi-structural responses and towards relational ones. In addition to aligning with the grading expectations of EiPE questions, such a feedback mechanism could be used to give students a sense of progression rather than simply giving them dichotomous feedback, which is often less effective than more granular feedback [14].

The second is a mechanism for helping students bridge the gap between their descriptions of code and how elements of those natural language descriptions ultimately manifest as code (Figure 5b). In terms of integrating this mechanism into the “Code Generation Based Grading” pipeline proposed by Smith and Zilles [23], by performing prompt segmentation *in addition to* code generation we could not only provide feedback on *correctness* and *high-levelness*, but also provide visual mappings between the segments of their prompt and their associated line(s) of code. Such feedback may be particularly valuable for helping students debug responses that might be categorized as *multi-structural error*, where a description

is multi-structural in nature but contains incorrect descriptions of one or more structures.

Design Affordances: The current output of the system described in this paper can be used to provide feedback by:

- Indicating to the student that they should reduce the number of highlighted segments of code.
- Map response segments to associated code to show how portions of a description manifest as code.

7 Limitations and Future Work

Though our results suggest the system is currently effective at performing segmentation insofar as it can be used to differentiate between multi-structural and relational responses, there remains the question of whether its approach to performing segmentation aligns with human expectations for what lines of code should be associated with which portions of the statement. Further, formalized analysis of the segments produced is needed before the mapping feedback shown in Figure 5b can be used reliably. Similarly, avenues for ensuring this reliability such as creating a mapping system between natural language and code that is both more formalized and generalizable across examples along with fine-tuning a model on such examples may prove useful in terms of ensuring the reliability of the system. Future work will consider such avenues towards further refinements.

8 Conclusion

The segmentation-based classification method introduced in this study demonstrates substantial promise in distinguishing between multi-structural and relational responses to code comprehension tasks. By prompting large language models with few-shot examples, the system achieves reasonable accuracy which was further improved through limited post processing of the resulting segments. Future work aims to refine the approach further through model fine-tuning and explore the effectiveness of using the resulting segments to provide feedback to students.

References

- [1] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.
- [2] 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* (Portland, OR, USA) (SIGCSE '20). Association for Computing Machinery, New York, NY, USA, 563–569. <https://doi.org/10.1145/3328778.3366879>
- [3] Binglin Chen, Matthew West, and Craig Zilles. 2022. Peer-grading "Explain in Plain English": A Bayesian Calibration Method for Categorical Answers. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) (SIGCSE 2022). Association for Computing Machinery, New York, NY, USA, 133–139. <https://doi.org/10.1145/3478431.3499409>
- [4] Tony Clear, Jacqueline Whalley, RF Lister, Angela Carbone, Minjie Hu, Judy Sheard, Beth Simon, and Errol Thompson. 2008. Reliably classifying novice programmer exam responses using the SOLO taxonomy. *National Advisory Committee on Computing Qualifications* (2008).
- [5] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
- [6] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.
- [7] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) (SIGCSE 2024). Association for Computing Machinery, New York, NY, USA, 296–302. <https://doi.org/10.1145/3626252.3630909>
- [8] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan. 2024), 56–67. <https://doi.org/10.1145/3624720>
- [9] Paul Denny, David H. Smith, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) (ITiCSE 2024). Association for Computing Machinery, New York, NY, USA, 283–289. <https://doi.org/10.1145/3649217.3653587>
- [10] Edsger W Dijkstra. 2005. On the foolishness of "natural language programming". *Program Construction: International Summer School* (2005), 51–53.
- [11] 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* (Virtual Event, USA) (SIGCSE '21). Association for Computing Machinery, New York, NY, USA, 1163–1169. <https://doi.org/10.1145/3408877.3432539>
- [12] 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* (Virtual Event, USA) (ICER 2021). Association for Computing Machinery, New York, NY, USA, 69–80. <https://doi.org/10.1145/3446871.3469738>
- [13] 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. <https://doi.org/10.1080/08993408.2022.2079866>
- [14] Qiang Hao, David H. Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H. Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (Jan. 2022), 105–127. <https://doi.org/10.1080/08993408.2020.1860408>
- [15] Chris Kerslake, Paul Denny, David H. Smith, James Prather, Juho Leinonen, Andrew Luxton-Reilly, and Stephen MacNeil. 2024. Integrating Natural Language Prompting Tasks in Introductory Programming Courses. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1* (Virtual Event, NC, USA) (SIGCSE Virtual 2024). Association for Computing Machinery, New York, NY, USA, 88–94. <https://doi.org/10.1145/3649165.3690125>
- [16] JR Landis. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* (1977).
- [17] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) (ITiCSE '09). Association for Computing Machinery, New York, NY, USA, 161–165. <https://doi.org/10.1145/1562877.1562930>
- [18] 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* (Sydney, Australia) (ICER '08). Association for Computing Machinery, New York, NY, USA, 101–112. <https://doi.org/10.1145/1404520.1404531>
- [19] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English' questions: implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) (SIGCSE '12). Association for Computing Machinery, New York, NY, USA, 385–390. <https://doi.org/10.1145/2157136.2157249>
- [20] James Prather, Brent N Reeves, Paul Denny, Juho Leinonen, Stephen MacNeil, Andrew Luxton-Reilly, João Orvalho, Amin Alipour, Ali Alfageeh, Thezyrie Amarouche, Bailey Kimmel, Jared Wright, Musa Blake, and Gweneth Barbre. 2025. Breaking the Programming Language Barrier: Multilingual Prompting to Empower Non-Native English Learners. In *Proceedings of the 27th Australasian Computing Education Conference* (ACE '25). Association for Computing Machinery, New York, NY, USA, 74–84. <https://doi.org/10.1145/3716640.3716649>
- [21] Victor-Alexandru Pădurean, Paul Denny, Alkis Gotovos, and Adish Singla. 2025. Prompt Programming: A Platform for Dialogue-based Computational Problem Solving with Generative AI Models. arXiv:2503.04267 [cs.CY] <https://arxiv.org/abs/2503.04267>
- [22] David H. Smith, Paul Denny, and Max Fowler. 2024. Prompting for Comprehension: Exploring the Intersection of Explain in Plain English Questions and Prompt Writing. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) (L@S '24). Association for Computing Machinery, New York, NY, USA, 39–50. <https://doi.org/10.1145/3657604.3662039>
- [23] David H. Smith and Craig Zilles. 2024. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) (ITiCSE 2024). Association for Computing Machinery, New York, NY, USA, 171–177. <https://doi.org/10.1145/3649217.3653582>
- [24] David H. Smith IV, Viraj Kumar, and Paul Denny. 2025. Explain in Plain Language Questions with Indic Languages: Drawbacks, Affordances, and Opportunities. , 3–17 pages. https://doi.org/10.1007/978-3-031-84391-4_1
- [25] 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* (Berkeley, CA, USA) (ICER '09). Association for Computing Machinery, New York, NY, USA, 117–128. <https://doi.org/10.1145/1584322.1584336>
- [26] Matthew West, Geoffrey L Herman, and Craig Zilles. 2015. Prairielearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. In *2015 ASEE Annual Conference & Exposition*. 26–1238.
- [27] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) (ACE '06). Australian Computer Society, Inc., AUS, 243–252.
- [28] 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.
- [29] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (March 2022), 47 pages. <https://doi.org/10.1145/3487569>