Leinonen, Juho; Hellas, Arto; Sarsa, Sami; Reeves, Brent; Denny, Paul; Prather, James; Becker, Brett A.

Using Large Language Models to Enhance Programming Error Messages

Published in:
SIGCSE 2023 - Proceedings of the 54th ACM Technical Symposium on Computer Science Education

DOI:
10.1145/3545945.3569770

Published: 02/03/2023

Document Version
Publisher's PDF, also known as Version of record

Using Large Language Models to Enhance Programming Error Messages

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

Brent Reeves
Abilene Christian University
Abilene, Texas, USA
brent.reeves@acu.edu

Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

James Prather
Abilene Christian University
Abilene, Texas, USA
james.prather@acu.edu

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

ABSTRACT

A key part of learning to program is learning to understand programming error messages. They can be hard to interpret and identifying the cause of errors can be time-consuming. One factor in this challenge is that the messages are typically intended for an audience that already knows how to program, or even for programming environments that then use the information to highlight areas in code. Researchers have been working on making these errors more novice friendly since the 1960s, however progress has been slow. The present work contributes to this stream of research by using large language models to enhance programming error messages with explanations of the errors and suggestions on how to fix them. Large language models can be used to create useful and novice-friendly enhancements to programming error messages that sometimes surpass the original programming error messages in interpretability and actionability. These results provide further evidence of the benefits of large language models for computing educators, highlighting their use in areas known to be challenging for students. We further discuss the benefits and downsides of large language models and highlight future streams of research for enhancing programming error messages.

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; **Computer science education**; • **Computing methodologies** → *Natural language generation*; **Artificial intelligence**.

KEYWORDS

AI; Codex; compiler error messages; large language models; programming error messages; syntax error messages

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569770

ACM Reference Format:
Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569770

1 INTRODUCTION

Programming Error Messages (PEMs) can be notoriously difficult to decipher, especially for novices [33], possibly to the extent that they contribute to the perception that programming is overly challenging [6]. Eye-tracking studies reveal that novices read error messages and spend a substantial amount of programming time trying to understand them [4]. Instructors report that they spend a considerable amount of time helping novices with these often cryptic messages [15, 31, 32, 37]. It is also known that error message presentation affects novice programming behavior [23]. For over six decades, researchers have attempted to improve these messages, and still there is a call for more work on the topic [7]. Some recent attempts have been made to put error messages into more natural language by focusing on an increase in readability [8, 17]. This has been shown to improve student understanding of error messages and the number of successful code corrections [16]. While it is clear that increasing the readability of PEMs is helpful to novices, doing so at scale, and across languages (both programming and natural [34]), remains a challenge.

Very recent work on using large language models in computing education has already produced promising results. One study found that Codex, a text-to-code model built on top of GPT-3, a text-to-text model (see Section 2.2) could solve introductory programming problems, and ranked Codex in the top quartile when compared to a cohort of actual students in a large introductory programming course [19]. Tools like Codex are able to generate new programming assignments [36] and code explanations [29] when provided examples. Such tools demonstrate the impressive interpretive power of very recent large language models that may

have the potential to improve the readability of input text. In this work, we investigate whether large language models can be utilized to parse non-compiling code and the programming errors generated from that code to output enhanced PEMs that are more readable than those generated by the compiler/interpreter.

The following research questions guide our analysis. RQ1: *How well can Codex explain different error messages?* RQ2: *What is the quality of the code fix suggestions that Codex generates?*

2 BACKGROUND

2.1 Programming Error Messages

Programming error messages (PEMs) encompass syntax error messages, compiler error messages, and other diagnostic messages that are produced by compilers or interpreters indicating that the input code violates the specification of a language [9]. Researchers and instructors have reported PEMs to be difficult for students since at least 1965 [35]. Nearly sixty years later, PEMs are still a barrier to progress for those learning to program [7], and this has led to various efforts to improve their usability.

One such avenue of work has involved intercepting messages between the compiler and the user and altering their wording or presentation. One of the many known issues with error messages generated by compilers and interpreters is lower than desired readability due to factors such as poor use of vocabulary, loose sentence structure, and the use of jargon [17]. Thus, a large body of work around so-called 'enhanced compiler error messages' has emerged [5, 9]. Different approaches to message wording have been reported by various authors, including Barik [3], Becker et al. [10], Denny et al. [14], Kohn [24], Pettit et al. [31], Prather et al. [33], and Ahmed et al. [2]. However, although some studies have shown positive effects of rewording messages for novices [9, 16], in general the evidence for the effectiveness of enhanced compiler error messages is not overwhelming [7]. One of the limitations of prior work in this area is the manual effort required to generate enhanced PEMs and a lack of clear guidance for addressing core issues such as readability [17].

Artificial intelligence and machine learning approaches have been used for finding and repairing errors in programs [1, 20, 21] but only very fundamental approaches have been applied to researching PEMs [11]. To our knowledge, no prior work has explored the use of large language models for improving PEMs.

2.2 Large Language Models

Large Language Models (LLMs), particularly pre-trained transformer models, have rapidly become a core technology in natural language processing [26]. One such model is GPT-3 (third-generation Generative Pre-trained Transformer) [12]. GPT-3 can translate between natural languages, compose poetry in the style of human poets, generate convincing essays, and more. GPT-3 also powers several other tools such as Codex which is essentially a GPT-3 model that has been fine-tuned with more than 50 million repositories representing the majority of Python code available on GitHub totalling 159 GB of source code [13]. Codex is available via the OpenAI API (beta.openai.com) and also powers tools such as GitHub Copilot (copilot.github.com).

Given the recent emergence of these models, little is yet known about the impact they are likely to have on the computing education landscape. In this context, the few evaluations conducted to date have focused on the accuracy of such models for solving typical introductory programming problems and on the potential for the models to generate learning resources. Early work by Finnie-Ansley et al. assessed the accuracy of Codex by presenting it with typical CS1-type problems, and comparing its performance against that of students. They found that it outperformed most students, and was capable of generating a variety of correct solutions to given problems [19]. Sarsa et al. investigated the content generation capabilities of Codex by providing input examples as prompts and using it to generate novel programming problems and code explanations [36]. They found that most of the problems generated by Codex were novel and sensible, and that the generated code explanations were generally correct and thorough.

Given the capability of large language models for generating output of human-like quality from contextual inputs, such as (natural-language) code explanations from code, we investigate the potential of using large language models in enhancing PEMs.

3 METHODOLOGY

3.1 Error Messages and Programs

For the present study, we collected Python error messages that had been reported as the most unreadable in [8] and [17]. These error messages were as follows:

(1) can't assign to function call
(2) invalid token
(3) illegal target for annotation
(4) unindent does no match any outer indentation level
(5) positional argument follows keyword argument
(6) unexpected EOF while parsing
(7) EOL while scanning string literal
(8) EOF while scanning triple-quoted string literal
(9) (unicodeerror) 'unicodeescape' codec can't decode bytes

To control whether the complexity of the program that results in a given error message affects the ability of large language models to create useful explanations of the message, we constructed three example programs that generated each error message. The first program was very simple, often only a few lines long. The second incorporated the usage of strings and functions. The third included the use of libraries (e.g., the PyGame game library, pandas, scikit-learn) and was more complex. To create the same error messages as in prior work [8, 17], we used Python version 3.6.

3.2 Generating Programming Error Messages

Programming error messages were generated using the most recent and performant Codex model available the time of analysis (`code-davinci-002` -model). As the utility of large language models depends on the prompts used as input (see e.g., [28]), it is important to consider "prompt engineering" where the performance of different types of prompts is evaluated [28]. We evaluated a number of prompts to identify a version that seemed to provide useful explanations. We tried five different prompt messages:

1. Plain English explanation of why does running the above code cause an error and how to fix the problem
2. Plain English explanation of why running the above code causes the above error in the output and instructions on how to fix the problem
3. Explanation of why running the above code causes the above error and instructions on how to fix the problem
4. Why does the code result in an error message? How can the code be fixed?
5. Why does the above code cause the above error message in the output? How can the code be fixed?

We generated explanations with all five prompts and assessed which version led to the fewest empty responses from Codex. The number of empty responses was 4, 6, 7, 16 and 27 out of 81 generated explanations respectively, for prompts 1 to 5 above. We chose the first for the analysis as it generated the fewest empty responses. The structure of the prompt given to the large language model can be seen in the Codex examples provided later in this article.

For each of the nine error messages, and each of the three programs leading to an error message, we generated three code explanations: one with the Codex temperature parameter set to 0, and two with the temperature set to 0.7. We chose these values as 0 is the minimum for the parameter and leads to least randomness, i.e., the most deterministic output. The value of 0.7 is the default and leads to more random (or 'creative') responses, and is less deterministic, which is also why we generated two explanations for the value of 0.7. This led to a total of $9 \times 3 \times 3 = 81$ unique combinations of programming error message, program category, and temperature value, which we subsequently evaluated.

3.3 Analysis

We qualitatively analyzed the LLM-produced PEMs. The evaluation was performed by two researchers, both of whom have experience teaching introductory programming. For the evaluation, we considered the following aspects of the generated PEMs.

(1) *Comprehensible*: was the generated content intelligible (i.e., proper English, not nonsensical)?
(2) *Unnecessary content*: did the generated explanation contain unnecessary content (e.g., repeating content, comprehensible but irrelevant content)?
(3) *Has explanation*: did the content produced by the LLM contain an explanation of the programming error message?
(4) *Explanation correct*: did the content produced by the LLM contain a *correct* explanation of the programming error message?
(5) *Has fix*: did the generated explanation contain actions or steps that one should take to fix the error?
(6) *Fix correct*: did the content produced by the LLM contain *correct* actions or steps that one should take to fix the error?
(7) *Improvement over the original*: did the explanation provide added value (from a novice programmer's standpoint) when compared to the original programming error message?

The researchers first discussed the aspects listed above to ensure a shared understanding of them, and jointly evaluated three examples. After the discussion and initial joint evaluation, they separately analyzed the full set of generated explanations. For each aspect, the researchers chose either "yes" or "no". For evaluation, the researchers also had access to the original error message as well as the program that produced the error message, and also considered these when evaluating the LLM-generated explanations. To assess the validity of the approach, we calculated inter-rater reliability using Cohen's kappa. The kappa value was 0.83 over all the analyzed aspects, indicating "almost perfect" agreement [25].

To answer our research questions, we report the percentage of "yes" answers for the questions outlined above separately for each programming error message, and for each combination of program category and temperature value. The proportion of "yes" answers is calculated out of the full set of 162 data points: 2 raters, each with 81 distinct ratings for the unique combinations of programming error message (n = 9), program (n = 3), and Codex output (n = 3).

4 RESULTS

Table 1 shows the results of the analysis separately for each error message. Each cell of the table presents the percentage of "yes" answers to the evaluation question (see Section 3.3 for the questions) for each of the nine error messages. The cells in the bottom row of the table show the percentage of "yes" answers across all error messages for the evaluation question indicated by the column.

In general, most error message explanations created by Codex were comprehensible (percentage of "yes" ranging from 67% to 100%). A few of the created explanations contained unnecessary content such as repeated sentences, extra question marks, etc. – the percentage ranging from 11% for "unexpected EOF while parsing" to 56% for "EOF while scanning triple-quoted string literal" and "(unicodeerror) 'unicodeescape' codec can't decode bytes".

In most cases, Codex successfully created an explanation of the error message (67% to 100% of the time depending on error message), although there were considerable differences between error messages on whether the explanation was correct. The range of correct explanations ranged from 11% for "unexpected EOF while parsing" to 83% for "can't assign to function call".

Regarding Codex's ability to create actionable fixes based on the faulty source code and the programming error message, we found that in the majority of cases, Codex provided a fix in the generated explanation (44% to 89% of cases). However, the fix was correct only 33% of the time, ranging from 17% of the time for "EOL while scanning string literal" to 56% for "(unicodeerror) 'unicodeescape' codec can't decode bytes".

Altogether, the evaluators considered that the Codex-created content, i.e., the explanation of the error message and the proposed fix, were an improvement over the original error message in slightly over half of the cases (54%). There were some differences between different error messages: the content was an improvement only 22% of the time for the "unexpected EOF while parsing" error message; while it was considered an improvement in 78% of the cases for "can't assign to function call" and "invalid token".

Table 2 shows the results of the analysis separately for different combinations of program category and temperature value. From the table, it is evident that for the task of explaining PEMs and creating suggestions for fixes to the source code that produced those errors, using a temperature value of 0 resulted in considerably better outputs, which holds for all three program categories. For

Error message	RQ1 Comprehensible	Unnecessary content	Has explanation	Explanation correct	Improvement	RQ2 Has fix	Fix correct
can't assign to function call	100%	17%	94%	83%	78%	72%	28%
invalid token	100%	39%	89%	50%	78%	83%	44%
illegal target for annotation	67%	22%	67%	33%	33%	50%	28%
unindent does not match any outer indentation level	100%	39%	100%	56%	56%	67%	28%
positional argument follows keyword argument	89%	22%	89%	61%	56%	78%	39%
unexpected EOF while parsing	67%	11%	67%	11%	22%	44%	22%
EOL while scanning string literal	89%	28%	89%	22%	50%	67%	17%
EOF while scanning triple-quoted string literal	89%	56%	78%	44%	44%	89%	33%
(unicodeerror) 'unicodeescape' codec can't decode bytes	89%	56%	83%	72%	67%	78%	56%
Average over all error messages	88%	32%	84%	48%	54%	70%	33%

Table 1: Error message analysis for each research question. The cells show the percentage of "yes" answers out of all ("yes" and "no") answers for the analysis.

Program category	Temperature	RQ1 Comprehensible	Unnecessary content	Has explanation	Explanation correct	Improvement	RQ2 Has fix	Fix correct
Simple	0.0	100%	6%	100%	67%	72%	78%	44%
Function with strings	0.0	100%	22%	100%	56%	72%	78%	33%
Library	0.0	100%	28%	100%	78%	78%	72%	44%
Simple	0.7	83%	31%	78%	47%	42%	64%	31%
Function with strings	0.7	89%	42%	86%	36%	39%	75%	25%
Library	0.7	72%	44%	64%	33%	50%	61%	31%

Table 2: Effect of temperature and program category on Codex performance in the task.

example, the output was considered an improvement in over 70% of the cases with a temperature value of 0, while only up to 50% of the cases with a temperature value of 0.7.

Regarding differences between program categories, we can observe that Codex seems to perform slightly worse with the programs in the "function with strings" category. However, the variations between program categories are not as noticeable as the differences between different temperature values or different error messages.

5 DISCUSSION

5.1 Are Error Message Explanations Useful?

Our results suggest that using large language models to explain programming error messages (PEMs) is feasible and shows promise. Overall, the explanation was considered an improvement over the original programming error message in over half of the cases. If we only consider the results from using a temperature value of 0, over 70% of the Codex outputs were considered an improvement over the original programming error message.

The results are more sobering when it comes to using large language models to generate correct fixes. Although 70% of the created content included a fix, when a fix was included it was correct only under half of the time (47%). While the fixes created with the temperature value of 0 were better on average compared to those created with a value of 0.7, they were still correct in only around half of the cases – 42%, 56%, and 61% of the cases for functions with strings, simple programs, and library related programs respectively.

We propose that the generated content could be useful for students if it was delivered so that it is clear that the content is AI-generated and might not be correct. And even with this initial exploratory setup, we found some outputs (two examples seen in Codex Example 1 and Codex Example 2) that we consider good enough to be shown to students without modifications.

Codex Example 1 (input in black, output in red)

```
""" Code
def check_password(password, input):
  return password == input
input = "hunter2"
if check_password("s3cr37", input):
""" Output
  File "main.py", line 5

                                      ^
SyntaxError: unexpected EOF while parsing
""" Plain English explanation of why does running
↪  the above code cause an error and how to fix the
↪  problem
The error is caused because the block of code is
↪  expecting another line of code after the colon.
To fix the issue, I would add another line of code
↪  after the colon.
```

5.2 Common Pitfalls and Ways Around Them

Two examples of outputs where both the explanation and suggested fix generated by Codex were incorrect are shown in Codex Examples 3 and 4. Comparing these incorrect outputs with the correct outputs in Codex Examples 1 and 2, we observe that the messages seem similarly confident in their tone, which could potentially mislead students. In both of the examples where the output is incorrect,

Codex Example 2 (input in black, output in red)

```
""" Code
users_dir_path = "C:\Users"
print("Users directory is", users_dir_path)
""" Output
  File "main.py", line 1
    users_dir_path = "C:\Users"
                     ^
SyntaxError: (unicode error) `unicodeescape' codec
↪  can't decode bytes in position 2-3: truncated
↪  \UXXXXXXXX escape
""" Plain English explanation of why does running
↪  the above code cause an error and how to fix the
↪  problem
The error is caused by the backslash character in
↪  the string.
The backslash character is used to escape characters
↪  that would otherwise have a special meaning,
↪  such as newline, backslash itself, or the quote
↪  character.
Since the backslash is used for escaping, if you want
↪  to use it in your string, you need to escape it.
You can do that by using a double backslash:
```

Codex suggests that the issue is related to indentation. As novices often struggle with indentation [24, 27], these incorrect suggestions could exacerbate this by potentially misleading students and even introduce misconceptions related to correct indentation.

In general, we observed a few common pitfalls that Codex seemed to often struggle with: (1) source code clearly missing a part of the content (resulting in "unexpected EOF while parsing", see Codex Example 3), (2) incorrectly capitalized control statements (resulting in "illegal target for annotation", see Codex Example 4), and (3) missing quotation marks (resulting in either "EOL while scanning string literal" or "EOF while scanning triple-quoted string literal").

For the first case, Codex would often suggest fixing the indentation of the program, even though the problem was that the implementation was far from complete (as in Codex Example 3). Similar suggestions for fixing the indentation were observed for the second case as well, even though the problem is in the capitalization. This can be seen in Codex Example 4, where the issue is that the if-statement is capitalized, but the message claims that the issue would be with the indentation. For the third case, Codex was often unable to correctly identify whether the quotation mark was missing from the beginning or the end of the string, and sometimes suggested that the issue is related to parentheses instead of missing quotation marks. Indeed, the program category "function with strings" had the lowest scores overall (see Table 2).

While it was relatively rare, we did observe some outputs that were not just incorrect, but even contradictory and confusing. In one case, Codex seems to have focused too much on the "Plain English" portion of the input and started generating irrelevant content related to "looking for a plain English explanation". To add to the confusion, the generated output actually does include a correct explanation of the problem – "You need to end your string with three single quotes at the end of your string to make it work", but the output also states that "this is not a correct explanation".

Codex Example 3 (input in black, output in red)

```
""" Code
import pygame

display = pygame.display.set_mode((640, 400))
pygame.display.set_caption("Awesome game!")

def main():
    while True:
""" Output
  File "main.py", line 8

                    ^
SyntaxError: unexpected EOF while parsing
""" Plain English explanation of why does running
↪  the above code cause an error and how to fix the
↪  problem
# The above code is not indented enough to be able to
↪  be run by Python.
# Add more spaces or tabs to the start of each line
↪  of code to be able to run it.
```

As there were common pitfalls and clear differences between explanation quality, we see one stream of future work in using a two-tiered approach for creating explanations. Codex could be relied upon in cases where it is known that it likely performs well, while in other cases other means could be exercised. One possibility is using LLMs to pre-generate explanations of common error messages that the instructor could validate (essentially, a "human-in-the-loop" approach). Another possibility would be the use of learnersourcing, where students could ask for help from their peers, or LLM-scaffolded learnersourcing [18]. Classic approaches such as discussion forums could also work, although response times would likely be lower when compared to the near-instantaneous feedback from a large language model.

5.3 Explanations and Context

When considering the usefulness of Codex-generated explanations, they need to be interpreted and evaluated in context. First, the original error messages might be more useful for more experienced students who have learned to interpret them. The importance of context was also present in some of the disagreements between the two researchers who independently evaluated the error messages; for example, one of the researchers at times considered the error message as an improvement if it pointed the students in the correct direction, even if the explanation by itself would be faulty.

The utility of these explanations also depends on if students understand the implications of the suggestions. Prior research investigating LLMs has shown that when used to facilitate the creation of source code, they may lead students down debugging rabbit-holes [38] or even introduce security flaws [30]. We also see the potential for other types of problems related to using LLMs to enhance PEMs. For example, what if the problem is not with the source

Codex Example 4 (input in black, output in red)

```
""" Code
def check_password(password, input):
  If (input == "s3cr37"): print("You are in!")
  Else: print("Wrong password!")
input = "hunter2"
check_password("s3cr37", input)
""" Output
  File "main.py", line 2
    If (input == "s3cr37"): print("You are in!")
    ^
SyntaxError: illegal target for annotation
""" Plain English explanation of why does running
↪ the above code cause an error and how to fix the
↪ problem
The error is caused by the fact that the code is not
↪ indented properly. The code should be indented
↪ by 4 spaces.
```

code, but an issue with the user environment? Here, a student could ask for help to fix an issue, convincing the LLM that an issue exists, and going down a rabbit-hole when looking for a solution [38].

Despite their shortcomings, we see the potential of using LLMs as a scaffold when learning to program and when learning to interpret error messages. However, as with any instructional scaffolding, the scaffolding should be dismantled at some point [22], and students must eventually learn to understand the original error messages.

5.4 Limitations

There are limitations to our study, which we outline here. Firstly, we used Python 3.6 in the analysis similar to prior work [8, 17]. On one hand, this allowed us to focus on error messages from the literature that had been found to be confusing to students. On the other hand, we acknowledge that newer versions of Python have included improvements to some of the error messages we analyzed. For example, some of the programs we used that resulted in an "invalid token" error would have resulted in a "SyntaxError: leading zeros in decimal integer literals are not permitted; use an 0o prefix for octal integers" with newer Python versions. We consider the latter to be easier to understand for novice programmers.

Regarding the programs analysed, they were created by the authors and were not student code. It is possible that the performance of Codex in explaining error messages for student code would be different. In future work, we are interested in studying error message explanations with student programs evaluators. In addition, most of the source codes were relatively short. The performance of large language models in explaining error messages might be affected by the length or the complexity of source code, which future work should examine in greater detail. Similarly, the programs only included singular errors. Future work could analyze how well large language models can explain error messages when the source code that leads to those messages contains multiple issues.

When prompting Codex to generate an explanation of the error message and a fix to the program, we asked for both the explanation and the fix with a single prompt ("Plain English explanation of why does running the above code cause an error and how to fix the problem"). Performance could have increased had we asked for these separately. In addition, we did not give any examples of good error message explanations and fixes to the code in the prompt – i.e., we relied on "zero-shot learning" [28]. Prior work has found that giving even just a few examples (i.e. "few-shot learning") can drastically improve the performance of large language models [12].

6 CONCLUSION

We used large language models to enhance programming error messages (PEMs). We collected Python error messages reported as most unreadable in prior work [8, 17] and generated code examples that produced these PEMs. We used prompt engineering with Codex [13] to identify prompts that would produce explanations of PEMs and actionable fixes that could be applied to the code examples to fix the error. We evaluated the explanations and fixes created to examine whether they have utility in introductory programming classrooms. To summarize, we answer our research questions.

RQ1: How well can Codex explain different error messages? Overall, the explanations created by Codex were quite comprehensible (88%). Codex produced an output with an explanation to 84% of the provided codes and error messages, but only about half (57%) of these explanations were deemed correct (48% of all inputs).

RQ2: What is the quality of the code fix suggestions that Codex generates? Although 70% of the outputs had a proposed fix, a little less than half (47%) of those were deemed correct (33% of all inputs).

While the above results are aggregated over different PEMs, program categories, and Codex temperature values, we found cases where Codex seems to perform better. For example, we noticed that the results were better across the board when using a temperature value of 0. Similarly, we found that there were certain cases where Codex was more likely to provide faulty explanations and suggest fixes that are incorrect, and highlighted a potential way around this by having a two-step system that would look into the error message and the complexity of the source code before deciding whether to use LLMs or other more traditional support mechanisms.

We find that PEM explanations and suggested fixes generated by LLMs are not yet ready for production use in introductory programming classes, as there are risks that students may interpret potentially faulty LLM outputs as coming from an authority, and attempt to fix their programs in ways that do not actually help. At the same time, our results show that LLMs could be a useful tool for enhancing PEMs, although additional effort needs to be taken both when using LLMs to enhance the error messages and when coming up with ways to produce high-quality enhancements. Enhancing programming error messages could help students in debugging their programs as traditional error messages are often cryptic and hard to understand for novice programmers [16, 17].

The present results were obtained with the `code-davinci-002` model of Codex, which was the most recent and performant Codex model at the time of the study. As LLMs improve over time, these results create a baseline that future model performance can be compared to. Future work should look in more depth into prompt engineering, for example, by considering including the problem statement, and perhaps including a sample solution in the input, as well as look into applying and evaluating the enhanced programming error messages in classroom settings.

REFERENCES

[1] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2021. SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics. *arXiv preprint arXiv:2104.14671* (2021).

[2] Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ACM, 78–87.

[3] Titus Barik. 2018. *Error Messages as Rational Reconstructions*. Ph. D. Dissertation. North Carolina State University.

[4] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do Developers Read Compiler Error Messages?. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 575–585.

[5] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. ACM, NY, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[6] Brett A. Becker. 2021. What Does Saying That 'Programming is Hard' Really Say, and About Whom? *Commun. ACM* 64, 8 (2021), 27–29.

[7] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ACM, 177–210.

[8] Brett A. Becker, Paul Denny, James Prather, Raymond Pettit, Robert Nix, and Catherine Mooney. 2021. Towards Assessing the Readability of Programming Error Messages. In *Australasian Computing Education Conference*. ACM, 181–188.

[9] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175.

[10] Brett A. Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 640–645.

[11] Brett A Becker and Catherine Mooney. 2016. Categorizing Compiler Error Messages with Principal Component Analysis. In *12th China-Europe International Symposium on Software Engineering Education (CEISEE 2016)*.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 19th Conference on Innovation and Technology in Computer Science Education*. ACM, 273–278.

[15] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM, 208–212.

[16] Paul Denny, James Prather, and Brett A Becker. 2020. Error Message Readability and Novice Debugging Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 480–486.

[17] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM.

[18] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources–Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).

[19] 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.

[20] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 930–937.

[21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common C Language Errors by Deep Learning. In *Thirty-First AAAI conference on artificial intelligence*.

[22] Slava Kalyuga. 2009. The Expertise Reversal Effect. In *Managing cognitive load in adaptive multimedia learning*. IGI Global, 58–80.

[23] Ioannis Karvelas, Annie Li, and Brett A. Becker. 2020. The Effects of Compilation Mechanisms and Error Message Presentation on Novice Programmer Behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, 759–765.

[24] Tobias Kohn. 2019. The Error Behind The Message: Finding the Cause of Error Messages in Python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 524–530.

[25] J Richard Landis and Gary G Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *biometrics* (1977), 159–174.

[26] Hang Li. 2022. Language Models: Past, Present, and Future. *Commun. ACM* 65, 7 (2022), 56–63.

[27] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 666–671.

[28] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *arXiv preprint arXiv:2107.13586* (2021).

[29] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating Diverse Code Explanations using the GPT-3 Large Language Model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*. 37–39.

[30] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[31] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students? Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 465–470.

[32] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.

[33] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 74–82.

[34] Kyle Reestman and Brian Dorn. 2019. Native Language's Effect on Java Compiler Errors. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, NY, NY, USA, 249–257. https://doi.org/10.1145/3291279.3339423

[35] Saul Rosen, Robert A. Spurgeon, and Joel K. Donnelly. 1965. PUFFT—The Purdue University Fast FORTRAN Translator. *Commun. ACM* 8, 11 (1965), 661–666.

[36] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*. 27–43.

[37] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4 (2013), 1–40.

[38] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–7.