Not the Silver Bullet: LLM-enhanced Programming Error Messages are Ineffective in Practice

Eddie Antonio Santos
School of Computer Science, University College Dublin
Ireland
eddie.santos@ucdconnect.ie

Brett A. Becker
School of Computer Science, University College Dublin
Ireland
brett.becker@ucd.ie

Abstract

The sudden emergence of large language models (LLMs) such as ChatGPT has had a disruptive impact throughout the computing education community. LLMs have been shown to excel at producing correct code to CS1 and CS2 problems, and can even act as friendly assistants to students learning how to code. Recent work shows that LLMs demonstrate unequivocally superior results in being able to explain and resolve compiler error messages—for decades, one of the most frustrating parts of learning how to code. However, LLM-generated error message explanations have only been assessed by expert programmers in artificial conditions. This work sought to understand how novice programmers resolve programming error messages (PEMs) in a more realistic scenario. We ran a within-subjects study with $n = 106$ participants in which students were tasked to fix six buggy C programs. For each program, participants were randomly assigned to fix the problem using either a stock compiler error message, an expert-handwritten error message, or an error message explanation generated by GPT-4. Despite promising evidence on synthetic benchmarks, we found that GPT-4 generated error messages outperformed conventional compiler error messages in only 1 of the 6 tasks, measured by students' time-to-fix each problem. Handwritten explanations still outperform LLM and conventional error messages, both on objective and subjective measures.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → *Compilers*; **Error handling and recovery**; • **Human-centered computing** → **Empirical studies in HCI**.

Keywords

AI; compiler error messages; computing education; CS1; debugging; feedback; GenAI; Generative AI; GPT-4; C; LLMs; large language models; novice programmers; PEM; programming error messages

ACM Reference Format:
Eddie Antonio Santos and Brett A. Becker. 2024. Not the Silver Bullet: LLM-enhanced Programming Error Messages are Ineffective in Practice. In *The United Kingdom and Ireland Computing Education Research (UKICER 2024), September 05–06, 2024, Manchester, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3689535.3689554

UKICER 2024, September 05–06, 2024, Manchester, United Kingdom

ACM ISBN 979-8-4007-1177-0/24/09
https://doi.org/10.1145/3689535.3689554

1 Introduction

For decades, students learning how to code have struggled with error messages [5]—whether they are emitted by compilers or runtime systems, programming error messages (PEMs) have had a reputation for being terse [2], inadequate [8], and unreadable [12]. Error messages from C and C++ compilers especially have been shown to be deficient debugging tools [43].

Recent advances in generative AI have resulted in tools like ChatGPT and GitHub Copilot. These tools, based on large language models (LLMs), have revolutionised several fields including computing education [4]. Recent work has shown that LLMs can produce acceptable programming error message explanations [22] which become more accurate with larger models and more source code context [39, 48]. However, it is unknown to what extent that novice programmers are able to effectively utilise these automatically generated error message explanations to debug their programs.

In this study, we had 106 students from an introductory programming module partake in a within-subjects study that had participants fix a number of buggy programs. For each program, participants were shown either a conventional compiler error message, an expert-handwritten error message, or an LLM-generated explanation. The LLM used to enhance error messages was GPT-4, which at the time of the study, was the LLM used in the paid version of ChatGPT [30]. We measured both how long it took participants to resolve errors with each condition, as well as asking students their opinions on their debugging experience.

1.1 Contributions & Research Questions

We provide empirical evidence demonstrating that students appear not to be any faster at resolving errors when given GPT-4 error message explanations compared to stock compiler error messages. Even though students are not any faster, they still prefer GPT-4's explanations to conventional compiler error messages. However, expert-handwritten are superior to both. We posit that error message usability is more complex than whether or not the text presented to the student contains the correct solution for the problem.

The following questions guide this research:

RQ1 How quickly do students resolve error messages when given LLM-enhanced explanations in comparison to stock compiler error messages and handwritten explanations?
RQ2 Which style of error message do students prefer?

2 Background and Related Work

2.1 Programming Error Messages

Programming error messages (PEMs) are the diagnostic messages presented to coders when an error is detected in their program—either due to a mistake in syntax or spelling in the source code, or due to some unrecoverable runtime condition, such as a division by zero, or an invalid memory access [5]. PEMs have been an obstacle to learning how to code since almost the inception of programming [2, 8, 47]. Little progress has been made to improve compiler and runtime error messages, despite decades of guidelines proposed to improve them [1, 12, 16, 19, 23, 43]. There have been many attempts at having programming systems produce better diagnostics [9, 20, 25, 37, 40], however, error message enhancement has seen weak [3, 6, 15, 34] to insignificant [11, 32] results in improving student outcomes.

2.2 Large Language Models and CS Education

A confluence of advances in model architecture; novel text representation; massive, curated datasets; and sheer computing power has rapidly enabled the development of large language models (LLMs) [36]: models with billions or even trillions of parameters, capable of capturing the structure and predictability of text to such an extent that they are able to exhibit "emergent" behaviours, like question answering, analogical reasoning, and even the ability to execute programs [45].

In computing education, LLM-powered tools have been shown to ace CS1 [13] and CS2 [14] exams and provide increasingly accurate error message explanations [22, 39, 48]. LLMs have even enabled brand new pedagogical approaches [10, 28]. Educators are grappling with how to integrate LLMs into their practice [4]—if at all [21]. Without guidance, complete novices struggle to write the prompts that would complete their assignments [28]. Additionally, novices exhibit a number of unproductive interaction patterns when using LLM-assisted code completion [35, 44]. Some programmers do not complete programming tasks faster with LLM-assisted code completion, and in fact, are more likely to fail programming tasks [44]. Having an LLM that performs better in synthetic benchmarks results in "relatively indistinguishable differences in terms of human performance" [27]. Despite the lack of improvement, novices express a preference for using LLMs and chatbots [31, 35, 44]. However, more experienced students express concern with how LLMs may hamper their learning [33].

LLMs and PEMs. LLMs have been found to be useful at explaining programming error messages on synthetic benchmarks. Leinonen et al. [22] used OpenAI Codex to explain Python error messages. They found that the best, most accurate explanations and fixes were obtained when providing source code in the prompt, as well as using a temperature value of 0 (explained in section 3.1). Their prompt forms the basis of the prompt that was used in our study. Santos et al. [39] and Widjojo and Treude [48] have similar findings: providing Java source code in the prompt produces significantly better error message explanations. Additionally, more advanced models like GPT-4 are more likely to output accurate explanations and fixes than GPT-3 and Codex.

3 Methodology

We conducted a within-subjects study, inspired by prior work [44]. Each participant observed all three study conditions—control, handwritten, and GPT-4 (Section 3.1). Each participant was tasked to fix all six buggy C programs (Section 3.2). Both condition assignment and task assignment were randomised to counterbalance responses, such that we would obtain a roughly equal amount of responses for each task/condition pair. Randomising participants' assignments also helped to mitigate the learning effect. Having participants fix bugs under all three study conditions allowed them to directly compare the different error message styles to one another, and report which style they preferred. The study began with a short questionnaire, after which participants were given an in-person briefing, which was followed by the six debugging tasks. After participants had completed all six tasks, we asked participants questions to compare the three error message styles directly. The remainder of this section describes the study conditions, the tasks, and study protocol in greater detail.

3.1 Study conditions

Each participant saw error messages presented in all three study conditions: **GCC** (control), **handwritten**, and **GPT-4**. However, the order of the study conditions was shuffled for each student, so that participants could directly compare the three study conditions themselves. Since there were six tasks, but only three conditions, the order of the study conditions was simply repeated for the latter three tasks—for example, if a participant was assigned handwritten for the first task, GCC for the second, and GPT-4 for the third, then they would be assigned handwritten for the fourth, GCC for the fifth, and GPT-4 for the sixth.

Control: GCC. For the control condition, students were presented with error messages directly obtained from the GCC 13.2.0 C compiler (Figure 1a). Whenever a single programming error would induce multiple spurious, cascading error messages, we would only show the first error message, as advised by Becker et al. [7].

Handwritten error messages. Error message explanations (Figure 1b) were handwritten by the first author. These explanations were written in response to the problems present in the source code, but not necessarily in response to any error messages emitted by GCC. Importantly, the handwritten explanations were finalised *before* error message explanations were obtained from GPT-4. Therefore, the author of the handwritten explanations was not influenced by GPT-4's output. Every message was written in a consistent structure: first was a line beginning with the word **Error:** which states what the detected error is, followed by a relevant excerpt from the source code. Then one or more sections beginning with the word **Help:** or **Note:** would either suggest a possible solution, or highlight relevant information to fix the problem. The structure of the messages was greatly inspired by the diagnostics emitted by the Rust compiler, with source code excerpts mimicking the structure of Rust's "diagnostic windows" [38]. The handwritten explanations were written in such a way that they can plausibly be generated by an actual compiler, given sufficient context.

GPT-4 enhanced error messages. After the handwritten error explanations were written, we obtained error message explanations

```
main.c:3:2: error: invalid preprocessing directive #const
```

(a) GCC (control)

Error: `#const` is not a valid preprocessor directive.

```
3 | #const pi 3.141592653589793
    ~~~~~~ invalid
```

Help: If you are trying to define a preprocessor constant, use `#define` instead:

```
3 | #define pi 3.141592653589793
    ^^^^^^^ change here
```

(b) Handwritten

Error Message:

```
main.c:3:2: error: invalid preprocessing directive #const
```

Explanation:

The error is caused because the `#const` directive is not a valid preprocessing directive in C. The preprocessor directives in C start with a `#` symbol, such as `#include`, `#define`, etc. However, `#const` is not one of them.

To define constants in C, you should use the `#define` directive or the `const` keyword. The `#define` directive is a preprocessor directive which is used to define symbolic constants, while the `const` keyword is used to declare variables whose value cannot be changed after initialization.

Here's how you can fix the problem:

```
#include <stdio.h>

#define PI 3.141592653589793

float area(float radius) {
    return PI * radius * radius;
}

int main() {
    printf("The area of the circle is: %.1f\n", area(1.128375));
    return 0;
}
```

In this corrected code, `#define PI 3.141592653589793` is used instead of `#const pi 3.141592653589793`. This defines `PI` as a symbolic constant that can be used throughout the code.

(c) GPT-4

Figure 1: Examples of the three error message styles (i.e., study conditions) as they would appear to participants.

using OpenAI's GPT-4 API. All GPT-4 responses were obtained before the study started, as to not include inference time in the time-to-fix measure.[1] The methodology for generating error message explanations is derived from prior work [22, 39]. On January 25, 2024, we used the OpenAI API to prompt GPT-4 model `gpt-4-0613`. Each prompt used the system message of "You are a helpful assistant". We prompted with a temperature of 0, a hyperparameter used to affect the determinism of an LLM's output, where 0 indicates the most deterministic and reproducible output.[2] The prompt used was identical to that in Santos et al. [39], providing both the error message verbatim from GCC as well as the full source code of the buggy program.

[1] Obtaining the full output for each of these prompts would take between 16–30 seconds, depending on the problem.

[2] A value greater than 0 introduces entropy into the sampling distributions, in a style reminiscent of Ludwig Boltzmann's work on thermodynamics.

When presenting the error message to participants (Figure 1c), the message would start with **Error Message:** followed by the GCC error message. This is because GPT-4's output would often make reference to the original error message in its explanations. After this, a line starting with **Explanation:** would be followed by GPT-4's output, rendered as Markdown. GPT-4 output would vary depending on the problem, however, every response consistently had a line saying something similar to "here's the corrected code:" followed by a full reproduction of the source code with the problem resolved. In all six problems, GPT-4 generated a solution equivalent to the one suggested in the handwritten explanations. In addition, GPT-4's error message explanations were devoid of any major technical inaccuracies or "hallucinations" [18].

3.2 Tasks

Students were to fix all of the following C programming errors, in a randomly assigned order. All programs in this study caused GCC to emit compiler error messages. We did not have students debug problems that would result in runtime errors (e.g., no segmentation faults). The following are all six of the debugging tasks:

(1) **Flipped assignment**. The left- and right-hand sides of an assignment statement were swapped such that the assignment target would be on the right-hand side, e.g., `a + b = c`.
(2) **`#const` instead of `#define`.** A program was made attempting to define a constant called `PI` using the syntax **`#const`** `PI 3.14`. This programming error intentionally conflates C's **`#define`** preprocessor directive with the **`const`** type qualifier, both being valid ways to define a constant in C.
(3) **Using a keyword as a name**. The program attempts to create new variables called **`union`** and `nonUnion`, however, **`union`** is a reserved word, and cannot be used as an identifier.
(4) **Missing parameter**. A function was defined to convert from Fahrenheit to Celsius, however, the function definition lacks any formal parameters. Despite this, the body of the function used a identifier `fahrenheit` to perform the conversion, and the function is called with an argument for the temperature in Fahrenheit.
(5) **Missing curly brace**. The opening curly brace of a function definition was omitted, causing GCC's parser to take a "garden-path" and completely misinterpret the program.
(6) **Reassigning a constant**. A formal parameter was declared **`const`**, then reassigned within the function body.

3.3 Participants

Participants were recruited from a class at a large research-intensive European public university that would be an R1 in the US Carnegie Classification. The class was the second semester of the first-year programming sequence (CS1) for CS majors, taught in the C programming language. In total, 113 participants were recruited across two separate lab sections. Of those, $n = 106$ participants (94%) completed the study. Of the participants that completed the study, 78 identified as men, 23 as women, and 5 chose not to disclose their gender. At the beginning of the study, we asked participants a few questions on their experience in programming. The most commonly reported experience level was between 0–3 months (32 participants). Curiously, 8 participants reported absolutely zero experience in

programming, despite being enrolled in the second semester of the programming sequence. The remaining 66 participants reported greater than three months of programming experience.

3.4 Protocol

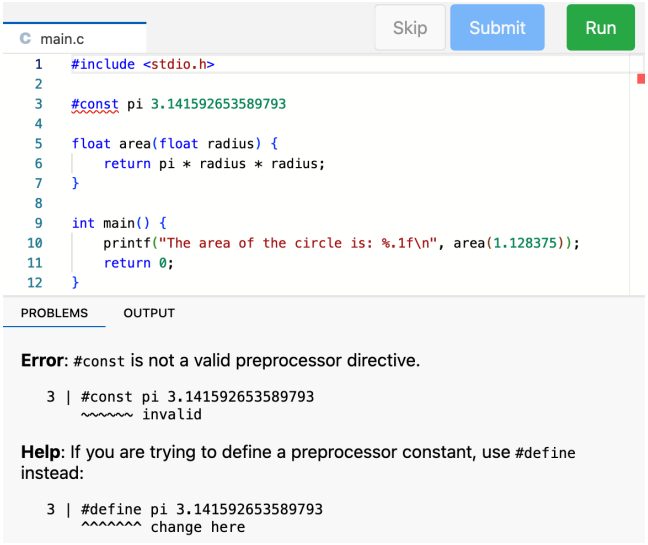

Figure 2: Screenshot of the web-based IDE, showing the "`#const` instead of `#define`" task under the handwritten error message condition.

The study was conducted during a regularly scheduled lab session in late January 2024—the second scheduled lab of the semester—held simultaneously in two separate classrooms. Students were under no obligation to participate in the lab session; they were not compensated for participation, and participation did not affect the students' grade in any way.

Once in the classroom, the study was conducted entirely via a custom web-based survey platform, which combined multiple questionnaires with an online IDE (Figure 2). The online IDE component was created using Microsoft's Monaco Editor, the core text editing component of Visual Studio Code [24]. Upon hitting the "Run" button in the IDE, the full source code was securely transmitted to a university-managed server, where it would be stored. Code was compiled and run in a sandboxed environment provided by the Piston code execution engine [41].

At the start of the study, the first author was present and gave a quick oral briefing. After the briefing, students were directed to the web platform, where they signed an online consent form to commence the study. First, participants were asked a few demographic questions and asked about their attitudes regarding programming and error messages. After this, participants started the six debugging tasks.

For each task, students were presented with the buggy code in the web IDE (Figure 2) and were instructed to fix the problem, hitting the "Run" button and only submitting their solution once the problem was resolved. We started measuring the time taken to complete the task from the moment the code editor loaded. After 5 minutes, students were given the option to skip the current task (if they were stuck on a problem and wanted to continue to the next one). Students were given up to 10 minutes per task, after which their attempt would have timed-out; however, all 106 participants either submitted or skipped each task before the 10 minute time-out would have taken effect. During the study, postgraduate TAs were instructed to ensure that students were not using AI tools like ChatGPT/GitHub Copilot. However, the students were still permitted to search the web for error message explanations and use Q&A sites like Stack Overflow. After each exercise, students were given a short questionnaire to reflect on how they used the error message to solve (or not solve) the programming error. This was presented as four Likert-type questions (Figure 4), followed by a question asking about the message's length. In total, 106 participants completed all six tasks within 50 minutes.

4 Results

4.1 Objective measures

The primary quantitative measurements that we recorded were the time-to-fix for participants who successfully fixed a task, and whether or not a student skipped a particular task. For each task/-condition pair, we obtained between between 27–44 samples, which is sufficient to perform within-task comparisons.

Table 1: Tukey's HSD test of the means of log-transformed time-to-fix, comparing the error message condition. The difference in median time-to-fix between the left and right condition is given. Bold font indicates the condition with the faster (statistically significant) time-to-fix.

Task	Comparison		Diff	p
#const instead of #define	GCC	**Handwritten**	43.95s	0.003
	GCC	GPT-4	4.93s	0.893
	Handwritten	GPT-4	-39.02s	0.002
Using a keyword as a name	GCC	**Handwritten**	63.42s	<0.001
	GCC	**GPT-4**	28.61s	0.006
	Handwritten	GPT-4	-34.82s	<0.001
Missing parameter	GCC	Handwritten	0.54s	0.716
	GCC	GPT-4	-15.08s	0.021
	Handwritten	GPT-4	-15.61s	0.157
Missing curly brace	GCC	**Handwritten**	122.3s	<0.001
	GCC	GPT-4	50.6s	0.373
	Handwritten	GPT-4	-71.6s	<0.001
Reassigning a constant	GCC	**Handwritten**	53.0s	0.031
	GCC	GPT-4	6.28s	0.996
	Handwritten	GPT-4	-51.6s	0.029

Time-to-fix. Time-to-fix is consistently right-tailed, so we log-transformed the data to perform statistical comparisons. We noticed that each task given had an effect on time-to-fix, so we performed within-task comparisons. For each task, we performed a one-way ANOVA to compare the effect of the study condition on students' log-transformed time-to-fix (Figure 3). A one-way ANOVA revealed that there was a statistically significant ($p < 0.05$) difference in the mean log-transformed time-to-fix between the conditions in all tasks, except for the "flipped assignment" task, in which no statistically significant difference was found ($p = 0.147$). We will

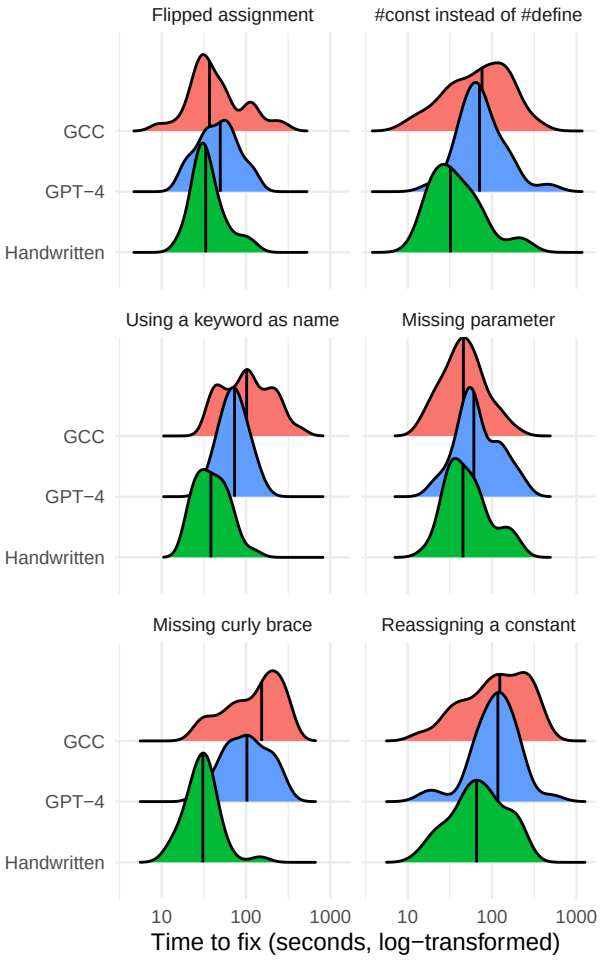

Figure 3: Density plots of the log-transformed time-to-fix, by task. The black vertical line denotes the median.

omit the flipped assignment task for the remainder of this section. For the five remaining tasks, we performed Tukey's HSD test on the means of log-transformed time-to-fix (Table 1). Because the mean of log-transformed time-to-fix is not useful to report, we instead report the difference in median time-to-fix, as median is preserved after log-transformation. We found that GPT-4 outperforms the control (GCC error messages) in only one of the five remaining tasks, namely, "using a keyword as a name". In three of the five tasks, we could not find a statistically significant difference in the mean log-transformed time-to-fix between GPT-4 and the control. However, the handwritten error messages outperformed both the control and the GPT-4 enhanced error messages in all tasks, except the "missing parameter" task, where we could not find a statistically significant difference between the handwritten error messages and the control; however, stock GCC error messages outperformed GPT-4's explanations.

Skip rate. We gave the option for students to skip exercises after spending 5 minutes on them without submitting a solution. Overall, students rarely skipped exercises, with only 13 skips out of 636 exercise attempts (2.0%). In other words, students successfully fixed the programming errors in 98% of all exercises. There were zero skips observed for students fixing tasks under the handwritten condition. The most skips (10) were observed for the control condition, whereas GPT-4 had only 3 skips. A χ^2-test failed to find a statistically significant difference between the three conditions.

4.2 Subjective measures

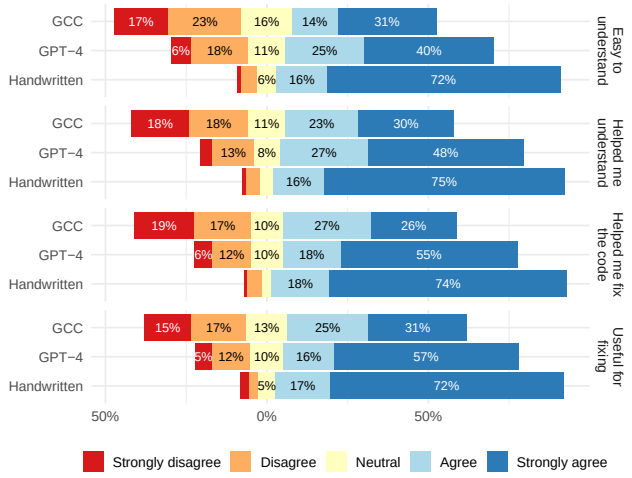

Figure 4: Proportion of participants' Likert responses. From top-to-bottom, the questions were "The message was easy to understand", "The message helped me understand what was wrong with the code", "The message helped me fix the code", and "The message was useful for fixing the problem".

Opinion. After each exercise, we asked participants four Likert-type questions to gauge their general opinion on how useful the message was for fixing the problem (Figure 4). We performed a linear regression to predict participants' overall opinion given the study conditions. We modelled opinion as a numerical variable where *Strongly disagree* = -2, *Disagree* = -1, *Neutral* = 0, *Agree* = 1, and *Strongly agree* = 2, then took the mean of a participant's responses per each condition. Using a handwritten message results in a 1.27 point increase in opinion compared to traditional compiler error messages ($p < 0.001$); whereas using a GPT-4 generated message results in a 0.69 point increase in opinion compared to the control ($p < 0.001$). Overall, we found that participants rated both the handwritten messages and GPT-4 explanations higher than GCC's error messages, with the handwritten error messages being the most highly rated. Participants rated GPT-4 error messages highly in terms of being useful to help them solve the error, despite both conditions suggesting equivalent fixes.

Message length. In addition to opinion, we also asked participants to rate the length of the error message after each exercise, on a scale from *way too short*, *too short*, *just right*, *too long*, to *way too long*. Participants overwhelmingly found that the handwritten error

messages were just the right length (88.1% of responses). GCC's error messages were mostly deemed as either the right length or too short—never way too long. GPT-4's error messages were roughly binomially distributed across all five categories.

5 Discussion

Despite promising evidence in prior studies [22, 39, 48], GPT-4 error message explanations do not help novices when they are resolving error messages as much as one would expect. In fact, in one of the tasks ("missing parameter"), students were *slower* when using GPT-4's explanation. Curiously, expert-handwritten error messages outperform GPT-4 error message explanations even though both suggested equivalent solutions for each problem.

When it comes to students' preferences, they preferred GPT-4's error messages over GCC's terse, jargon-heavy error messages. This makes sense, as GPT-4 would always produced full, complete sentences—a factor that was previously found to be important to error message readability [12]. We were surprised that participants did not report GPT-4's error messages as being too long. That said, students were unable to use these messages effectively, even though GPT-4's messages would always provide the correct way to solve the programming error. Prior work has found that longer error messages do not seem to help students [29].

Programming is (still) hard. Early results in understanding LLMs' capabilities at introductory programming seemed promising [13, 14, 22, 39], inspiring researchers to declare a new era for computing education [4] and even "the end of programming" [46]. However, it seems that LLMs are not the transformative tool that they once seemed. Our findings—that, despite excelling in synthetic benchmarks, LLMs do not significantly improve programmers' productivity—are corroborated by a number of studies [17, 27, 28, 44]. Additionally, participants express preference for LLMs [35, 44], even though LLMs' answers do not make them more effective at resolving programming errors. Simkute et al. [42] argue that this supposed contradiction of LLM productivity is an already well-known phenomenon in human factors research: automation alters peoples' workflows in unproductive ways, such as turning active producers into passive evaluators. This change in workflow widens the gap between the programmer's mental model, and what the programmer *ought* to attend to while solving problems. In fact, there is evidence that LLM-powered code suggestion alters programmers' workflows in ways that hamper productivity [26, 35]. LLMs have not delivered on the promise of natural language programming [28]; rather, they provide an indirect method of manipulating an existing abstraction: high-level source code. Similarly, LLMs have not fundamentally altered the task of debugging; they just explain the already difficult problem in a more approachable manner. Thus, debugging remains just as difficult as it was prior to the introduction of LLMs.

5.1 Limitations

The generalisability of these results is limited by the sample of participants: all from one class at a European research university, taught in one programming environment. Additionally, since the debugging tasks were created for the purpose of this study, the programming errors may be inauthentic to the kind of debugging that would naturally occur during programming. Another limitation was that the same person who prepared the debugging tasks also wrote the handwritten error messages. There is also the confound in that the GPT-4 error messages always presented the stock compiler error message verbatim, rather than produce their own explanation of the error, without using the original error message as context. It is possible that participants read the stock compiler error message, without reading further to use the GPT-4 explanation.

6 Conclusion

We conducted a within-subjects experiment where novice programmers fixed six buggy programs using three different error message styles. In contrast to results on synthetic benchmarks, GPT-4 enhanced error messages were only more effective than stock compiler error messages in one of the six programming tasks. Handwritten error messages were more effective than the control in four of six tasks, and more effective than GPT-4's error message explanations in five of six tasks. Overall, students preferred GPT-4's messages over stock compiler error messages, but preferred the handwritten error messages even more. This is despite the fact that the GPT-4 error message explanations and the handwritten error messages both made the same suggestion to fix the problem. Future work should further understand what factors make an error message usable and how programming environments and LLMs alike can be modified to satisfy novices' needs. It appears we still have a long way to go to reach "the end of programming".

Acknowledgments

Thanks to Dennis Bouvier for inspiration on creating the debugging tasks; Gavin McArdle and Di Meng for graciously helping us run the study in their lab sessions; Simon Caton, Kira Finan, and Olivia Finan for help with data analysis; and Sajjad Karimian, Fionn Murphy, and Abdul Wadud for beta testing the online IDE. This study received approval from our institutional ethics review board (LS-23-56-Santos-Becker).

References

[1] Titus Barik. 2018. *Error Messages as Rational Reconstructions.* Ph. D. Dissertation. North Carolina State University.
[2] D. W. Barron. 1975. A Note on APL. *Comput. J.* 19, 1 (1975), 93.
[3] 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.
[4] 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* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, NY, NY, USA, 500–506.
[5] 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* (Aberdeen, Scotland, UK) *(ITiCSE-WGR '19)*. ACM, NY, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508
[6] 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* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 640–645.
[7] Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney, and Kate Sanders. 2018. Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, NY, NY, USA, 634–639. https://doi.org/10.1145/3159450.3159453

[8] P. J. Brown. 1983. Error Messages: The Neglected Area of the Man/Machine Interface. *Commun. ACM* 26, 4 (Apr 1983), 246–249.

[9] CJ Burgess. 1972. Compile-time error diagnostics in syntax-directed compilers. *Comput. J.* 15, 4 (1972), 302–307.

[10] 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)*. ACM, New York, NY, USA, 296–302.

[11] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. ACM, NY, NY, USA, 273–278. https://doi.org/10.1145/2591708.2591748

[12] 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* (Yokohama, Japan) *(CHI '21)*. ACM, NY, NY, USA, Article 55, 15 pages.

[13] 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 *Proceedings of the 24th Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. ACM, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[14] 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 *Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. ACM, NY, NY, USA, 8 pages. https://doi.org/10.1145/3576123.3576134

[15] Devon Harker. 2017. *Examining the Effects of Enhanced Compilers on Student Productivity*. Master's thesis. University of Northern British Columbia.

[16] James J. Horning. 1974. What the Compiler Should Tell the User. In *Compiler Construction: An Advanced Course*, F. L. Bauer, F. L. De Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, W. M. Waite, F. L. Bauer, and J. Eickel (Eds.). Springer, Berlin, Heidelberg, 525–548.

[17] Brij Howard-Sarin. 2024. The Future of the Error Message: Comparing Large Language Models and Novice Programmer Effectiveness in Fixing Errors. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2* (Portland, OR, USA) *(SIGCSE 2024)*. ACM, New York, NY, USA, 1881.

[18] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. arXiv:2311.05232 [cs.CL]

[19] 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*. Association for Computing Machinery, New York, NY, USA, 524–530. https://doi.org/10.1145/3287324.3287381

[20] Tobias Kohn and Bill Manaris. 2020. Tell Me What's Wrong: A Python IDE with Error Messages. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, NY, NY, USA, 1054–1060.

[21] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 106–121.

[22] 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* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, NY, NY, USA, 563–569. https://doi.org/10.1145/3545945.3569770

[23] Caleb Meredith. 2019. *Writing Good Compiler Error Messages*. Code → Software. Retrieved August 24, 2022 from https://calebmer.com/2019/07/01/writing-good-compiler-error-messages.html

[24] Microsoft. 2024. *Monaco Editor*. Retrieved April 10, 2024 from https://microsoft.github.io/monaco-editor/

[25] P. G. Moulton and M. E. Muller. 1967. DITRAN—a compiler emphasizing diagnostics. *Commun. ACM* 10, 1 (jan 1967), 45–52. https://doi.org/10.1145/363018.363060

[26] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. arXiv:2210.14306 [cs]

[27] Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers. arXiv:2404.02806 [cs.SE]

[28] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. arXiv:2401.15232 [cs.HC]

[29] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler error messages: what can help novices? *SIGCSE Bull.* 40, 1 (Mar 2008), 168–172. https://doi.org/10.1145/1352322.1352192

[30] OpenAI. 2023. GPT-4 System Card. Retrieved May 14, 2023 from https://cdn.openai.com/papers/gpt-4-system-card.pdf

[31] Bruno Pereira Cipriano and Pedro Alves. 2024. "ChatGPT Is Here to Help, Not to Replace Anybody" — An Evaluation of Students' Opinions On Integrating ChatGPT In CS Courses. arXiv:2404.17443 [cs.ET]

[32] 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* (Seattle, Washington, USA) *(SIGCSE '17)*. ACM, New York, NY, USA, 465–470.

[33] Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. 2023. Generating Programs Trivially: Student Use of Large Language Models. In *Proceedings of the ACM Conference on Global Computing Education Vol 1* (Hyderabad, India) *(CompEd 2023)*. ACM, New York, NY, USA, 126–132.

[34] 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* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 74–82.

[35] 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. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (Nov 2023), 31 pages. https://doi.org/10.1145/3617367

[36] Marco Ramponi. 2023. *The Full Story of Large Language Models and RLHF*. Retrieved June 5, 2024 from https://www.assemblyai.com/blog/the-full-story-of-large-language-models-and-rlhf/

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

[38] Rustc developers. 2024. *Errors and Lints — Rust Compiler Development Guide*. Retrieved March 4, 2024 from https://rustc-dev-guide.rust-lang.org/diagnostics

[39] Eddie Antonio Santos, Prajish Prasad, and Brett A. Becker. 2023. Always Provide Context: The Effects of Code Context on Programming Error Message Enhancement. In *Proceedings of the ACM Conference on Global Computing Education Vol 1* (Hyderabad, India) *(CompEd 2023)*. ACM, New York, NY, USA, 147–153.

[40] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* (Nashville, Tennessee, USA) *(SIGCSE '95)*. ACM, New York, NY, USA, 168–172.

[41] Brian Seymour and contributors. 2023. *Piston: A high performance general purpose code execution engine*. Retrieved April 10, 2024 from https://github.com/engineer-man/piston

[42] Auste Simkute, Lev Tankelevitch, Viktor Kewenig, Ava Elizabeth Scott, Abigail Sellen, and Sean Rintel. 2024. Ironies of Generative AI: Understanding and mitigating productivity loss in human-AI interactions. arXiv:2402.11364 [cs.HC]

[43] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010 (2010), 26 pages.

[44] 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*. ACM, NY NY, USA, 1–7.

[45] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. arXiv:2206.07682 [cs.CL]

[46] Matt Welsh. 2022. The End of Programming. *Commun. ACM* 66, 1 (Dec 2022), 34–35. https://doi.org/10.1145/3570220

[47] Richard L. Wexelblat. 1976. Maxims for Malfeasant Designers, or How to Design Languages to Make Programming as Difficult as Possible. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 331–336.

[48] Patricia Widjojo and Christoph Treude. 2023. Addressing Compiler Errors: Stack Overflow or Large Language Models? arXiv:2307.10793 [cs.SE]