Synthetic Students: A Comparative Study of Bug Distribution Between Large Language Models and Computing Students

Stephen MacNeil
Temple University
Philadelphia, PA, US
stephen.macneil@temple.edu

Magdalena Rogalska
Temple University
Philadelphia, PA, US
m.rogalska@temple.edu

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

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

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

Xandria Crosland
xcrosl1@wgu.edu
Western Governors University
Millcreek, Utah, USA

Abstract

Large language models (LLMs) present an exciting opportunity for generating synthetic classroom data. Such data could include code containing a typical distribution of errors, simulated student behaviour to address the cold start problem when developing education tools, and synthetic user data when access to authentic data is restricted due to privacy reasons. In this research paper, we conduct a comparative study examining the distribution of bugs generated by LLMs in contrast to those produced by computing students. Leveraging data from two previous large-scale analyses of student-generated bugs, we investigate whether LLMs can be coaxed to exhibit bug patterns that are similar to authentic student bugs when prompted to inject errors into code. The results suggest that unguided, LLMs do not generate plausible error distributions, and many of the generated errors are unlikely to be generated by real students. However, with guidance including descriptions of common errors and typical frequencies, LLMs can be shepherded to generate realistic distributions of errors in synthetic code.

CCS Concepts

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

Keywords

Generative AI, LLMs, GPT-4, synthetic data, student data, buggy code, educational data mining

ACM Reference Format:
Stephen MacNeil, Magdalena Rogalska, Juho Leinonen, Paul Denny, Arto Hellas, and Xandria Crosland. 2024. Synthetic Students: A Comparative Study of Bug Distribution Between Large Language Models and Computing Students. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1 (SIGCSE Virtual 2024), December 5–8, 2024, Virtual Event, NC, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649165.3690100

SIGCSE Virtual 2024, December 5–8, 2024, Virtual Event, NC, USA

ACM ISBN 979-8-4007-0598-4/24/12
https://doi.org/10.1145/3649165.3690100

1 Introduction

Large language models (LLMs) present a promising new opportunity for generating synthetic data [20], which may have important implications for computing education research and practice. Such data could include examples of code containing typical student errors, which could have useful practical applications. For example, it provides a viable solution for conducting research in situations where access to authentic data is restricted due to privacy concerns [26]. Teachers could also make use of such examples when developing learning resources, such as debugging tasks, or materials to help address common mistakes. Having access to large amounts of synthetic data could also be useful for developers of educational tools. For example, a well known problem when developing intelligent or adaptive tutoring systems is the 'cold start problem' [38], where a lack of user data results in an initial mismatch between the system's internal model and a learner's actual performance.

Prior work outside of computing education contexts has begun to explore the use of LLMs for generating synthetic data. In the field of Human-Computer Interaction (HCI), Hämäläinen et al. generated synthetic questionnaire responses, demonstrating that LLMs can produce believable accounts of user experiences [21]. Although they show the potential for using synthetic data to ideate and pilot experiments, they suggest synthetic data should be validated with real data to ensure reliability. Similarly, Park et al. studied LLM-based agents to simulate human behavior and found interactions of the agents were human-like, and useful for designers [39, 40].

Inspired by these developments in other research areas, in this work we investigate the potential of LLMs to generate synthetic code that mimics the distribution of bugs found in student-written code. The central research questions of this study are:

- **RQ1:** Can LLMs produce erroneous code upon request?
- **RQ2:** To what extent does directing an LLM through prompt engineering influence the distribution of bugs it generates?
- **RQ3:** How do bug distributions from an LLM correlate with or deviate from those generated by human students?

To address these questions, we compare the distribution of bugs generated by an LLM with those produced by computing students. Using publicly-available student data from previous studies, we investigate the effectiveness of different prompting strategies in guiding LLMs to generate realistic error distributions. Our findings suggest that while LLMs do not produce accurate distributions

without guidance, they can do so with appropriate information about error frequencies. Our work is the first to explore the use of LLMs for generating synthetic data for computing education research purposes, and we discuss avenues for future work.

2 Related Work

2.1 Common Bugs in Students' Code

Learning to program involves developing an understanding of the syntax, structure and style of a programming language [33], and students encounter a wide range of syntax and logic errors during this process [2, 12, 42]. Such errors include "trivial mechanics" errors such as syntax errors with braces, brackets, semicolons, and naming conventions [42], as well as errors related to the semantics of the learned language [2]. Early research on errors in programming often centered on specific problems [25, 46, 48]. Later, researchers increasingly used programming errors – and programming process data – in forming a deeper understanding of the problems [23]. As an early example of such work, Jadud [24] quantified the error fixing behavior of novice programmers, identifying a link between the error fixing behavior (or skill) and course outcomes.

In general, there are differences in the frequency of programming errors [42, 49] and the time that it takes to fix such errors [8, 11]. The types of errors that students encounter also gradually change over time [2, 52], and they can stem from multiple sources [2, 16]. These sources include misinterpreting the programming problem and having flaws in programming knowledge [16], as well as the role of the programming language and the environment [28, 50, 52].

Research on programming errors has contributed to programming language design (e.g. [47]), and researchers have sought to help students with programming errors, for example, by improving programming error messages [4, 14, 30]. All such research builds on the availability of relevant data. However, although there are increasing number of programming datasets available [23], errors can vary between programming languages, and the distributions of errors may also differ between contexts.

2.2 Generating Educational Content With LLMs

Large language models (LLMs), which are advanced transformer models, possess the ability to both comprehend and generate code and text. These capabilities enable LLMs to offer students personalized, just-in-time pedagogical support [13] . For instance, LLMs have been utilized to enhance code comprehension by explaining code in plain English [29, 35, 43] or by producing analogies to explain both code and underlying concepts like recursion [5, 6]. Leinonen et al. found that the explanations provided by LLMs were comparable in word length to those generated by peers in a classroom, though students rated the quality of LLM-generated explanations higher [29]. However, in other cases, LLMs have been shown to dramatically outperform students such as in identifying bugs [34].

In addition to supporting students with just-in-time content generation, some research has explored the creation of real-time content for instructors. For example, Tran et al. demonstrated that LLMs can generate multiple-choice questions with plausible distractors and correct answers based solely on the question stem [51]. Doughty further extended this research by developing a pipeline for creating multiple-choice questions aligned with Bloom's Taxonomy [15]. In their work, they observed that the pipeline resulted in similar learning objectives as those covered in the class.

What this prior work shows is that LLMs already appear to be capable of generating effective learning materials. In some cases, these materials are comparable to the ones that might be sourced from students [29] or crafted by instructors [15, 51]. However, what remains unclear is how well these 'similar' materials can be aligned to mimic the style and performance of students, including the common errors they might make.

2.3 Synthetic Data Generation

There are many reasons for generating synthetic data for research. Real data could be scarce [17], hard to collect [21], low quality [17], or contain private information that cannot be shared (for example, medical records) [9]. One way of mitigating these issues is to try de-identify datasets [19], although this can in some cases reduce the utility of the data [31]. These issues can be potentially sidestepped by generating synthetic data, if the quality of the generated data is similar to or greater than organic data.

Traditional approaches for generating synthetic data have included algorithm-based approaches [44] and generative adversarial networks (GANs) [17]. In education, synthetic data has been used, for example, to generate data to train and evaluate knowledge tracing methods [44] which aim to accurately model a learner's knowledge of the concepts they are practicing.

Recently, advancements in large language models (LLMs) have opened new possibilities for synthetic data generation. Research has explored various scenarios, including simulating social interactions. Park et al. discovered that interactions within a simulated community, termed 'social simulacra,' can aid in prototyping, with simulated content often hard for experienced moderators to distinguish from real content [40]. In a follow-up study, they found that data generated by 25 LLM-based agents in a game-like environment was more believable than data from human crowdworkers [39]. Similarly, Hämäläinen et al. used GPT-3 to generate synthetic data on HCI experiences, particularly video games as art, finding the content human-like but less diverse than that created by humans [21].

3 Methods

To evaluate our research questions, we conducted two comprehensive studies. In the first study, we aimed to replicate and extend the findings of Altadmri and Brown [2]. Using their methodology, we employed GPT-4, a state-of-the-art LLM, to generate synthetic bugs. To address a lack of information about the precise programming problems used in their work, we conducted a second study where the programming problems were more explicitly defined. This second study replicates the bug frequencies identified by Rigby et al. [41] in their study of 900 programming students. An overview of the programs we used can be found in the virtual appendix[1].

3.1 Prompting With Distributional Information

To generate synthetic bugs, we investigated three prompting strategies that differed in the level of information provided about how

[1] https://figshare.com/s/e0b0db319ca9ef73fa0c

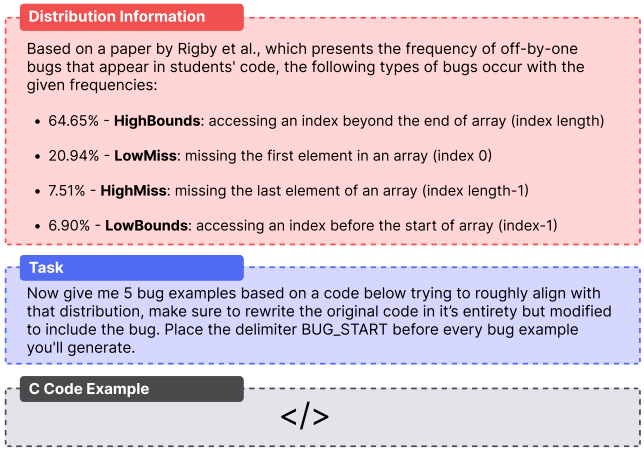

Figure 1: An overview of the prompts used in Study 2. The *Distributional Information* was used in the *Frequency-informed* and *Taxonomy-informed* prompts. However, in the *Taxonomy-informed* prompts, the specific frequency percentages were removed. The *Task* information and Code Example were used across all three prompts.

students encounter similar problems in real-world contexts. An example based on Study 2 is presented in Figure 1.

3.1.1 Open-Ended Prompt. The baseline prompt was intentionally open-ended to gauge how closely LLMs align with student bug frequency. Here, no specific information about bug distribution was provided to the model.

3.1.2 Taxonomy-Informed Prompt. In the second prompt, we explored the impact of providing basic information about the latent bug distribution. The model was given a list of bugs encountered by students in real-world contexts as reported by the two papers we replicated (i.e.: Altadmri and Brown [2] and Rigby et al. [41]). This approach reflects how an instructor might have some intuition about common student bugs without precise frequency knowledge.

3.1.3 Frequency-Informed Prompt. In the final prompt, we provided the model with detailed information about the latent distribution of bug frequencies, including the specific frequencies at which students encountered each error. We used or computed the frequencies for each study based on the two papers being replicated [2, 41].

3.2 Study 1: Replication of Altadmri and Brown

In the first study, we conducted a replication of the work by Altadmri and Brown [2]. Their research analyzed the frequency of bugs generated by real students across 37 million compilations. They identified 18 errors related to syntax, type, and semantics. These errors are summarized in Table 1.

To replicate their work, we chose five Java programs. Java was chosen because that was the language used by participants in their study [2]. The five Java programs were chosen to be diverse; however, they do not perfectly match the range used in the prior study where the programming problems were not experimentally controlled and varied widely across the 37 million compilations.

Table 1: Student Mistakes from Altadmri and Brown [2]

Shorthand	Explanation of the Mistake
Syntax errors:	
A	Confusing = with ==
C	Mismatched parentheses
D	Confusing & with &&
E	Spurious semi-colon after `if`, `for`, `while`
F	Wrong separator in `for`
G	Wrong brackets in `if`
H	Using reserved keywords
J	Forgetting parentheses when calling methods
K	Spurious semi-colon after method header
L	Less-than / greater-than operators wrong
P	Including types in actual method arguments
Type errors:	
I	Calling method with wrong types
Q	Type mismatch when assigning method result
Other semantic errors:	
B	Using == to compare strings
M	Invoking instance method as static
N	Discarding method return
O	Missing return statement
R	Missing methods when implementing interface

We used GPT-4 to generate bugs and experimentally varied the prompts as described in Section 3.1. In total, 375 output programs with injected bugs were created. This was done by requesting the generation of five bugs for five code examples and repeating this process five times for each of the three prompt permutations to account for the probablistic nature of LLM responses, resulting in:

`3 prompts x 5 trials x 5 code examples x 5 bugs`

3.3 Study 2: Replication of Rigby et al.

The goal of the first study was to investigate whether LLMs are capable of generating similar distributions of bugs and syntax errors as students. However, it was challenging to replicate this prior work because they had investigated bug frequencies extracted from thousands of authentic programs which were solving a great variety of programming tasks.

To more tightly control the programming tasks, we conducted a second study that replicates the work of Rigby et al. [41]. In their work, only four programming problems were used and the corresponding bug frequencies were computed based on more than 22,000 submissions from 900 students. They focused purely on logic errors, in particular off-by-one errors for C code that iterates over an array. They categorised the four mistakes that can cause an off-by-one error: missing the first element (index 0), missing the last element (index length-1), accessing an invalid index before the start (index -1), or accessing an index just past the end (index length).

Similar to Study 1, we used GPT-4 to generate bugs for the programming problems. We only modified the distributional information to align with the corresponding bugs and frequencies.

3.4 Analyzing the LLM-Generated Bugs

3.4.1 Deductive Coding and Inter-Rater Reliability. The first part of the analysis focused on deductively coding the generated data using the original taxonomies from each corresponding study [2, 41]. Two coders independently coded the data and we computed inter-rater reliability (IRR) using Cohen's Kappa to determine their agreement

Table 2: Comparison of bug frequencies (%) with Altadmri and Brown [2]. The table includes out-of-distribution bugs from our thematic analysis and '-' denotes refusals.

Bug	Bug Type	Original	Frequency	Taxonomy	Baseline
C	Syntax	33.1	17.6	13.6	4.8
I	Type	19.4	19.2	9.6	0
O	Semantic	14.3	17.6	12.8	4.8
A	Syntax	7.3	13.6	13.6	8.8
N	Semantic	5.1	5.6	2.4	0.8
B	Semantic	5.1	0.8	4.0	0.8
M	Semantic	3.6	4.0	2.4	1.6
R	Semantic	3.3	0	2.4	0
P	Syntax	2.2	0	4.0	0
E	Syntax	2.1	8.0	11.2	0.8
K	Syntax	1.6	0.8	3.2	0.8
D	Syntax	1.2	1.6	7.2	0
J	Syntax	0.8	0	0	0
Q	Type	0.7	0	0	1.6
L	Syntax	0.2	0	0.8	3.2
F	Syntax	0.1	0	0.8	0
H	Syntax	<0.1	0	0	0
G	Syntax	<0.1	0	0	0
Out-of-Distribution Errors					
-	None	N/A	1.6	5.6	3.2
X	Mixed	0	4	0.8	12.8
Y	Semantic	0	0	0.8	19.2
T	Type	0	2.4	0	5.6
W	Syntax	0	0	0	7.2
S	Semantic	0	0.8	0	18.4
U	Mixed	0	2.4	4.8	5.6

which is adjusted for class imbalances. For Study 1, these codes are listed in Table 1 and the Kappa for IRR between the two coders was 0.92. For Study 2, we used the same four bug types coded in their study, 'High Bounds', 'Low Bounds', 'Low Miss', and 'High Miss'. The Kappa for IRR between coders was 0.81.

3.4.2 Statistical Analyses. The resulting frequency data was analyzed using the Chi Square goodness of fit test. Given that there were multiple distributions being compared, we corrected the critical p-values using the Bonferroni correction, which reduces Type I errors due to multiple comparisons.

3.4.3 Thematic Analysis of Out-of-Distribution Bugs. When replicating both studies [2, 41], bug types that were not included in the original were coded 'X'. These data were then analyzed using a thematic analysis approach to identify additional themes. The thematic analysis was guided by best practices [7] and followed a multi-step process with two coders analyzing the data independently but frequently discussing what they were observing and mediating their understanding.

4 Results

4.1 Study 1

4.1.1 Generating Bugs With LLMs. Our results suggest that LLMs can effectively create and integrate bugs into otherwise correct code. The refusal rates across prompts were extremely low (3.46%) and this included instances where it returned correct code. Otherwise, the models were capable of producing and injecting bugs into the code. We did observe that the models often explicitly identified the bug in the code with a comment describing the bug.

Table 3: The themes of out-of-distribution bug types identified in our thematic analysis.

Y	*Logic error:* Examples include counting 0 and 1 as primes, a function returning false in an if statement where it should return true, or forgetting to use a temporary variable while swapping variables.
T	*Type error:* A function or variable has the wrong type. For example, a function with return type `int` returns a string, or a variable of type `double` is assigned a char value.
W	*Undeclared or uninitialized variables:* Trying to use an undeclared variable or modify a variable that has not been assigned a value.
S	*Off-by-one error:* Starting a loop at 1 instead of 0, or iterating past the valid address in an array.
U	*Operator confusion:* Confusing operators such as `%, /, =, &&, ||`. For example, using `if (element && toCheckValue)` instead of `if (element == toCheckValue)`.

4.1.2 Bug Frequencies by Prompt Type. Our results also show that providing the model with information about the distribution helped to ensure the distribution more closely matched the actual distribution of bugs generated by students. As shown in Table 2, the *Baseline Prompt*, which had no information about the bug types or associated frequencies, produced code containing 68.8% of bugs that were not in the original distribution. Conversely, the *Frequency-informed* and *Taxonomy-informed* prompts produced 9.6% and 6.4% of these out-of-distribution bugs.

Based on a Chi Square Test, we observed that the *Taxonomy-informed* ($\chi^2 = 8.7$, $p < 0.05$) and *Baseline* ($\chi^2 = 17.7$, $p < 0.01$) prompts produced distributions that were statistically significantly different than the distributions present in the students' code. However, there was no significant difference for the *Frequency-informed* prompt ($\chi^2 = 0.18$, $p = 0.91$). This suggests that the additional context was helpful in reproducing the original distribution.

Finally, across all three prompts, we observed a bias in the model where some bugs, such as A, E, and D were amplified by the model. For example, *Bug E*, which is the error of adding a semi-colon after an if, for, or while statement, was common for both the *Frequency-informed* (8.0%) and *Taxonomy-informed* (11.2%) prompts despite being uncommon in the original student distributions (2.1%)

4.1.3 Thematic Analysis of Out-of-Distribution Bugs. In our initial coding, out-of-distribution bugs were labeled *X*. We conducted a thematic analysis on these bugs to identify what types of bugs GPT-4 injected into the code. We identified five themes and additional bugs that did not fit into those themes. The five themes are described in Table 3. The frequency of these five bugs are also reported in Table 2. Many of these errors were not compilation errors and were therefore not reported in the original study [2].

4.1.4 Examples of Out-of-Distribution Bugs. In addition to the themes from the previous section, we also observed some 'X' bugs that were less common but very interesting. For instance, when using the *Baseline Prompt*, GPT-4 occasionally mixed syntax from different languages, such as confusing `.length()` with `.length`. Similarly, GPT-4 sometimes used `boolean` and `bool` interchangeably, and even misspelled it as `'bolean.'` These typos and syntactic confusions might reflect the types of errors students make when transitioning between programming languages [10].

Some examples of out-of-distribution bugs were ones that would indicate considerable confusion if made by students (and thus might

make useful teaching examples). For example, consider the following error produced from the *Taxonomy-informed Prompt*:

```
return nstr = ""; //Returning an assignment
of empty string
```

This bug is unusual because it conflates concepts such as return statements and variable assignments. Furthermore, assigning a variable and then immediately returning that variable is an unnecessary step as the empty string could just be returned directly. Moreover, in some languages this might return the memory address of the variable `nstr` (though in Java it would return the empty string) or produce a compilation error. Typically, programmers are taught to distinguish between assigning a value to a variable and returning a value from a function.

The *Taxonomy-informed* and *Frequency-informed* prompts seldom produced strange bugs. However, in one case the *Frequency-informed* prompt included a string in the method signature:

```
public static String reverse("Hello") {
```

Finally, there was an instance in the *Baseline* prompt where the model produced a conditional statement with the condition missing.

```
if () { return true; }
```

Like the previous example, this error is unusual because it does not serve a functional purpose. Where other errors, such as using syntax from different languages interchangeably are mistakes students might make, it is more difficult to understand why a student would make such an error.

4.1.5 Refusal Rates. While not common, we observed all three prompts resulted in the model refusing to add bugs to the code. The refusal rate was highest for the *Taxonomy-informed* prompt (5.6%) and lowest for the *Frequency-informed* prompt (1.6%). These differences are minor and likely driven largely by chance.

4.2 Study 2

The first study showed that providing information about the underlying distribution helped GPT-4 to replicate that distribution. However, certain types of programming problems are more prone to specific bugs. For instance, off-by-one errors are highly unlikely in code that does not involve iteration. As a result, not being able to use the same programming problems as Altadmri and Brown [2] was a limitation. To address this limitation, Study 2 focused on replicating prior work where only four programming problems (all consisting of iterating over an array) were attempted by every student in the study.

4.2.1 Bug Frequencies by Prompt Type. Similar to Study 1, the *Baseline* prompt produced more out-of-distribution errors (44%) than the *Frequency-informed* and *Taxonomy-informed* prompts which only contained 6% and 8% of bugs that were not in the original distribution respectively. Unlike Study 1, the *Baseline* prompt produced fewer out-of-distribution errors. This may be because the *Baseline* prompt for Study 2 was constrained by only asking for 'off-by-one' errors. We observed statistically significant differences between each distribution and the distribution of students' bugs. Based on Chi Square Tests (with p-values corrected using the Bonferroni correction), the *Frequency-informed* ($\chi^2 = 69.9$, $p < 0.01$), *Taxonomy-informed* ($\chi^2 = 115.7$, $p < 0.01$), and *Baseline* ($\chi^2 = 76.4$, $p < 0.01$) prompts produced distributions of off-by-one errors that were different to those seen in practice from real students. Manual inspection of the frequencies in Table 4 indicate that the *Frequency-informed* prompt produced a somewhat more realistic distribution, better matching the most common 'HighBounds' error type.

Table 4: Comparison of bug frequencies (%) with Rigby et al. [41] where '-' and 'X' represent refusals and out-of-distribution errors respectively.

Bug	Original	Frequency	Taxonomy	Baseline
HighBounds	64.6	30	21	20
LowMiss	20.9	21	21	13
HighMiss	7.5	19	21	0
LowBounds	6.9	21	20	19
LM and HB	0	2	2	1
LM and HM	0	0	2	0
Out-of-Distribution Errors				
-	N/A	1	5	3
X	0	6	8	44

4.2.2 Examples of Out-of-Distribution Off-By-One Errors. We observed in the data many interesting errors that were injected by the LLM which were not strictly 'off-by-one' errors (in the sense of loop iteration) but which did involve an adjustment (by 1) of a value or variable in the code. For instance, we observed bugs where the accumulator was decremented (or incremented) prior to it being returned by the function

```
return count-1;
```

We also observed some unusual errors, such as using a post-decrement operator within the loop condition, leading to quite subtle bugs (and which would cause the loop to terminate earlier than a typical off-by-one):

```
for (int i = 0; i < n--; i++){
```

4.2.3 Refusal Rates. Refusals were uncommon (1%–5%) and similar to Study 1, with *Frequency-informed* decreasing from Study 1 by 0.6%, *Taxonomy-informed* decreasing by 0.6% and *Baseline* decreasing by 0.2%.

5 Discussion

In this paper, we explored whether LLMs can generate realistic synthetic bugs – that is, that mirror the distribution of real bugs produced by students when working on programming problems. Our results indicated that providing the model with some guidance helped considerably to align the generated bugs to those observed in practice. In particular, providing a list of common bugs (*Taxonomy-informed*) tended to improve the generated distribution over not including this information, whereas including frequency information as well (*Frequency-informed*) provided even closer alignment. In Study 1, we found that including frequency information helped the model to produce a corpus of buggy code with a distribution of errors that was statistically similar to the original data.

In both studies, we used the OpenAI API when making requests to the GPT-4 LLM, thus relying on the frequency information provided in our prompts across independent API calls. In contrast, a chat-based LLM such as ChatGPT could take a more sophisticated

approach by generating code (to select values at random according to a distribution) and executing it with its underlying code interpreter plug-in. This would allow ChatGPT to produce a perfectly aligned distribution, limited only by its ability to generate correct bugs of each specified type, which is seems very capable of doing. Of course, in practice, it may not be possible to accurately determine the probabilities (of bugs, or any other artefacts to be synthesized) in advance. Even with data from prior research, as we had, such probabilities can be highly contextual and may vary from one programming task or educational setting to the next. For example, a previous replication of the original study by Brown and Altadmri with human students resulted in a slightly different distribution [1]. Nevertheless, our results demonstrate that LLMs have potential for generating synthetic data, such as bugs, using probability information when it is made available.

Almost all of the 18 categories of errors in Study 1, and all four of the logic error categories in Study 2, had matching bug examples generated by the LLM. This is quite impressive considering that the vast majority of code used in training LLMs is typically free from bugs, as code committed to public repositories is usually debugged beforehand. Thus, the syntax and logic errors made by novices when learning to program are likely not well represented in the data available to LLMs when training. However, especially for the baseline condition, some of the bugs produced were uncommon and in some cases unconventional. These bugs described as case studies in this paper were much more common for the baseline prompt which further highlights the value of including guidance to the models. Similarly, and consistent with prior work [3, 27, 37], we observed biases in LLMs where some uncommon bugs were amplified by the model, even when given the frequency.

5.1 Toward Synthetic Students

We see exciting avenues for future work exploring the use of LLMs to simulate individual students. For example, very recent work has shown that leveraging LLMs to simulate students answering MCQs can support item evaluation and help educators improve question quality [32]. This suggests great potential for running simulations involving synthetic students. A class of synthetic students, with different capability and error-proneness, could complete a proposed assessment to give feedback to the instructor on its suitability. In theory, it may even be possible to to test interventions using synthetic students, allowing for experimentation in a controlled, risk-free environment before applying them in actual classroom settings. We elaborate on these possibilities in the following subsections:

5.1.1 Cold-Start Problem. The cold-start problem [45] occurs when intelligent tutoring systems and autograders lack sufficient data to effectively assist students. This issue arises because these systems rely heavily on historical data to generate accurate recommendations and feedback. LLMs can help to supplant this need or to augment historical data if it is not sufficient.

5.1.2 Piloting In-Class Interventions. A persistent challenge in computing education research is the ethical concern of conducting interventions that might inadvertently harm students. One potential solution is to simulate student interactions within a classroom environment before implementing interventions. This approach allows researchers to test and refine their methods in a controlled, risk-free setting. While this strategy requires significant development and validation, it holds promise for improving the ethical standards of classroom research. Nonetheless, it is crucial that simulated approaches complement, rather than replace, in-class research involving actual students, as real-world student behavior cannot be fully captured by simulations. Simulations become another tool alongside pre-registration, informed consent, and power analysis.

5.1.3 Predictors of Success. Building on prior computing education research focusing on predicting the success of students based on early performance [18, 22], one possible direction is to train LLM agents based on students in a class and then use those agents to identify students that are at risk of failing by simulating their performance through the rest of the course.

Generating bugs similar to those encountered by students can also be beneficial for training TAs and instructors. This approach has been explored in the context of simulating students' responses to multiple-choice questions (MCQs) [32] and training TAs by creating LLM agents that ask them questions about the assignments [36]. Building on these efforts, LLMs could generate common bugs and coding design patterns, aiding TAs in identifying and addressing gaps in their knowledge before they begin working with students.

6 Limitations

There are a few limitations to consider in this study. First, as mentioned in the discussion, there may be more deterministic methods for replicating the distribution of student data. The goal in this work was to understand the impact that information about the distribution has on model alignment. This is important because precise distributions are not always known and can vary based on the student population and course context [1, 2, 52]. This work also highlights that using LLMs to generate bugs without providing any information about an expected distribution will result predominately in irrelevant bugs.

7 Conclusions

In this paper, we investigated the capabilities for LLMs to produce bugs with the same distribution as students in a classroom study. Across our two studies, we observed that giving the model information about the underlying distribution improved the ability of GPT-4 to produce relevant bugs. In cases where the distribution was not provided, GPT-4 was more likely to produce out-of-distribution bugs that in some cases would likely provide limited pedagogical benefit for students. Consequently, we propose the idea of synthetic students, which can mimic real student errors and behaviors, offering new opportunities for teacher training and practice.

References

[1] Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. 2018. Learning programming, syntax errors and institution-specific factors. In *Proceedings of the 20th Australasian computing education conference*. 90–96.

[2] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proc. of the 46th ACM Technical Symp. on Computer Science Education*. 522–527.

[3] Lena Armstrong, Abbey Liu, Stephen MacNeil, and Danaë Metaxa. 2024. The Silicone Ceiling: Auditing GPT's Race and Gender Biases in Hiring. *arXiv preprint arXiv:2405.04412* (2024).

[4] 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.

[5] Seth Bernstein, Paul Denny, Juho Leinonen, Lauren Kan, Arto Hellas, Matt Littlefield Sami Sarsa, and Stephen MacNeil. 2024. "Like a Nesting Doll": Analyzing Recursion Analogies Generated by CS Students using Large Language Models. *arXiv preprint arXiv:2403.09409* (2024).

[6] Seth Bernstein, Paul Denny, Juho Leinonen, Matt Littlefield, Arto Hellas, and Stephen MacNeil. 2024. Analyzing Students' Preferences for LLM-Generated Analogies. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2*. 812–812.

[7] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[8] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Trans. on Computing Education (TOCE)* 17, 2 (2017), 1–21.

[9] Fida K Dankar and Mahmoud Ibrahim. 2021. Fake it till you make it: Guidelines for effective synthetic data generation. *Applied Sciences* 11, 5 (2021), 2158.

[10] Paul Denny, Brett A. Becker, Nigel Bosch, James Prather, Brent Reeves, and Jacqueline Whalley. 2022. Novice Reflections During the Transition to a New Programming Language. In *Proc. of the 53rd ACM Technical Symp. on Computer Science Education - Vol. 1*. ACM, New York, NY, USA, 948–954.

[11] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proc. of the 17th ACM annual Conf. on Innovation and technology in computer science education*. 75–80.

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

[13] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable Characteristics for AI Teaching Assistants in Programming Education. *arXiv preprint arXiv:2405.14178* (2024).

[14] 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.

[15] Jacob Doughty, Zipiao Wan, Anishka Bompelli, Jubahed Qayum, Taozhi Wang, Juran Zhang, Yujia Zheng, Aidan Doyle, Pragnya Sridhar, et al. 2024. A comparative study of AI-generated (GPT-4) and human-crafted MCQs in programming education. In *Proc. of the 26th Australasian Computing Education Conf.* 114–123.

[16] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proc. of the 20th Australasian Computing Education Conf.* ACM, New York, NY, USA, 83–89.

[17] Alvaro Figueira and Bruno Vaz. 2022. Survey on synthetic data generation, evaluation methods and GANs. *Mathematics* 10, 15 (2022), 2733.

[18] Sally Fincher, Anthony Robins, Bob Baker, Ilona Box, Quintin Cutts, Michael de Raadt, Patricia Haden, John Hamer, et al. 2006. Predictors of success in a first programming course. In *Proc. of the 8th Australasian Computing Education Conf.*

[19] Simson Garfinkel. 2015. *De-identification of Personal Information:*. US Department of Commerce, National Institute of Standards and Technology.

[20] Xu Guo and Yiqiang Chen. 2024. Generative AI for Synthetic Data Generation: Methods, Challenges and the Future. arXiv:2403.04190 [cs.LG]

[21] Perttu Hämäläinen, Mikke Tavast, and Anton Kunnari. 2023. Evaluating Large Language Models in Generating Synthetic HCI Research Data: a Case Study. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '23)*.

[22] Arto Hellas, Petri Ihantola, Andrew Petersen, Vangel V. Ajanovski, Mirela Gutica, Timo Hynninen, Antti Knutas, Juho Leinonen, Chris Messom, and Soohyun Nam Liao. 2018. Predicting academic performance: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 175–199.

[23] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. *Proc. of the 2015 ITiCSE on working group reports* (2015), 41–63.

[24] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proc. of the second int. workshop on Computing education research*.

[25] W. Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. *Bug Catalogue: I.* Technical Report. Yale University, YaleU/CSD/RR #286.

[26] Kyle M. L. Jones, Andrew Asher, Abigail Goben, Michael R. Perry, Dorothea Salo, Kristin A. Briney, and M. Brooke Robertshaw. 2020. "We're being tracked at all times": Student perspectives of their privacy in relation to learning analytics in higher education. *J. of the Association for Information Science and Technology* 71, 9 (2020), 1044–1059.

[27] Hannah Rose Kirk, Yennie Jun, Filippo Volpin, Haider Iqbal, Elias Benussi, Frederic Dreyer, Aleksandar Shtedritski, and Yuki Asano. 2021. Bias out-of-the-box: An empirical analysis of intersectional occupational biases in popular generative language models. *Advances in neural information processing systems* 34 (2021).

[28] Tobias Kohn. 2019. The error behind the message: Finding the cause of error messages in python. In *Proc. of the 50th ACM Technical Symp. on Computer Science Education*. 524–530.

[29] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Proc. of the 2023 Conf. on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*.

[30] 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 *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 1*. 563–569.

[31] Juho Leinonen, Petri Ihantola, and Arto Hellas. 2017. Preventing keystroke based identification in open data sets. In *Proc. of the Fourth (2017) ACM Conference on Learning @ Scale*. 101–109.

[32] Xinyi Lu and Xu Wang. 2024. Generative Students: Using LLM-Simulated Student Profiles to Support Question Item Evaluation. *arXiv preprint arXiv:2405.11591* (2024).

[33] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, et al. 2018. Introductory programming: a systematic literature review. In *Proc. companion of the 23rd annual ACM conf. on innovation and technology in computer science education*. 55–106.

[34] Stephen MacNeil, Paul Denny, Andrew Tran, Juho Leinonen, Seth Bernstein, Arto Hellas, Sami Sarsa, and Joanne Kim. 2024. Decoding Logic Errors: A Comparative Study on Bug Detection by Students and Large Language Models. In *Proceedings of the 26th Australasian Computing Education Conference*. 11–18.

[35] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proc. SIGCSE'23*. ACM, 6 pages.

[36] Julia M Markel, Steven G Opferman, James A Landay, and Chris Piech. 2023. GPTeach: Interactive TA training with GPT-based students. In *Proceedings of the tenth acm conference on learning@ scale*. 226–236.

[37] Roberto Navigli, Simone Conia, and Björn Ross. 2023. Biases in large language models: origins, inventory, and discussion. *ACM Journal of Data and Information Quality* 15, 2 (2023), 1–21.

[38] Maciej Pankiewicz. 2021. Assessing the Cold Start Problem in Adaptive Systems. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2*. 650.

[39] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[40] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2022. Social simulacra: Creating populated prototypes for social computing systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–18.

[41] Liam Rigby, Paul Denny, and Andrew Luxton-Reilly. 2020. A miss is as good as a mile: Off-by-one errors and arrays in an introductory programming course. In *Proceedings of the twenty-second australasian computing education conference*.

[42] Anthony Robins, Patricia Haden, and Sandy Garner. 2006. Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 165–173.

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

[44] Sami Sarsa, Juho Leinonen, Arto Hellas, et al. 2022. Empirical Evaluation of Deep Learning Models for Knowledge Tracing: Of Hyperparameters and Metrics on Performance and Replicability. *Journal of Educational Data Mining* 14, 2 (2022).

[45] Andrew I Schein, Alexandrin Popescul, Lyle H Ungar, and David M Pennock. 2002. Methods and metrics for cold-start recommendations. In *Proc. of the 25th annual int. ACM SIGIR conf. on Research and development in information retrieval*.

[46] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proc. of the 15th Koli Calling Conf. on Computing Education Research*. 87–96.

[47] Elliot Soloway, Jeffrey G. Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.

[48] Elliot Soloway, Kate Ehrlich, Jeffrey G. Bonar, and Judith Greenspan. 1982. What do novices know about programming? In *Directions in Human–Computer Interactions*. Vol. 6. Ablex Publishing, 27–54.

[49] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[50] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4 (Nov. 2013).

[51] Andrew Tran, Kenneth Angelikas, Egi Rama, Chiku Okechukwu, David H Smith, and Stephen MacNeil. 2023. Generating multiple choice questions for computing courses using large language models. In *2023 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[52] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Proc. of the 14th Koli Calling Int. Conf. on Computing Education Research*. 109–116.