Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants

Gustavo Sandoval[*], Hammond Pearce[*], Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt
New York University

Abstract

Large Language Models (LLMs) such as OpenAI Codex are increasingly being used as AI-based coding assistants. Understanding the impact of these tools on developers' code is paramount, especially as recent work showed that LLMs may suggest cybersecurity vulnerabilities. We conduct a security-driven user study (N=58) to assess code written by student programmers when assisted by LLMs. Given the potential severity of low-level bugs as well as their relative frequency in real-world projects, we tasked participants with implementing a singly-linked 'shopping list' structure in C. Our results indicate that the security impact in this setting (low-level C with pointer and array manipulations) is small: AI-assisted users produce critical security bugs at a rate no greater than 10% more than the control, indicating the use of LLMs does not introduce new security risks.

1 Introduction

Large Language Models (LLMs) are deep neural networks trained on massive text corpora [1, 2] to learn the underlying distribution of natural language or structured text. When trained on code, LLMs can be used for code completion, bug fixing, and summarization [3–5], useful features for developers. Recent offerings are thus commercializing LLMs for code, including GitHub Copilot, which after its public release in June 2022 added 400,000 new users in just two months [6].

However, recent work has shown that LLM completions may contain critical security vulnerabilities [7, 8]. This suggests that despite gain in developer productivity, LLM based code assistants should be used with caution (or not at all) due to security concerns. This prior work has evaluated the security of LLM code assuming that its entirely generated by the LLM (we will call this the autopilot mode). In practice, code completion LLMs *assist* developers with suggestions that they can accept, edit or reject—a real-world security evaluation must account for the role of developers and how they interact

[*]Equal Contribution

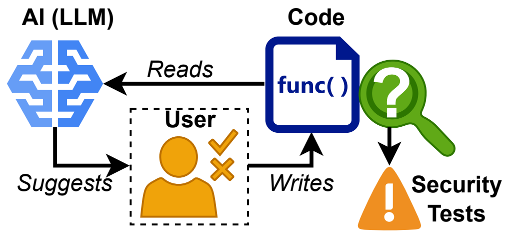


Figure 1: What is the security impact of LLM assistance?

with LLM based code assistants. While programmers prone to automation bias might naively accept buggy completions, other developers might produce overall less buggy code by only accepting safe suggestions and using time saved to fix other bugs.

This leads us to the key question motivating this work: *Do developers with access to an LLM-based code completion assistants produce less secure code than the code produced by programmers without this access* (Fig. 1)? An affirmative answer to this question could be a significant showstopper for LLM based code assistants. To answer this question, we perform the *first* security-motivated randomized trial comparing programmers with and without access to a Codex-based code completion assistant powered by OpenAI's `code-cushman-001` LLM.

Our user study in Section 3 had 58 computer science undergraduate and graduate students with programming backgrounds split randomly into 'control' (no Codex LLM access) and 'assisted' (with Codex LLM access) groups. Given the relative frequency of memory-based errors in low-level languages such as C and C++ ($\approx$70 % of CWEs assigned by Microsoft each year [9]), as well as their relative severity (classes of memory related bugs take many of MITRE's 'Top 25 Common Weakness Enumeration (CWE) Most Dangerous Software Weaknesses' list, including positions #1 and #5 [10]), we design a study where the participants were (t)asked to complete a set of 12 functions that perform basic operations on a linked list representing a "shopping list" in C. To understand the programming patterns in both groups, we created a cloud-based integrated development environment (IDE) that links to a Codex LLM in the back-end to behave like GitHub Copilot. The IDE logs user inputs and interactions with Codex LLM at a fine-grain. Using data from the user study, we investigate the impact of LLMs across three research questions:

RQ1: Confirm motivation for this research: Does an AI code assistant help novice users write better functional code?

RQ2: Given functional benefits, does the code that users write with AI assistance have an acceptable incidence rate of security bugs vis-a-vis code written without assistance?

RQ3: How do AI assisted users interact with potentially vulnerable code suggestions—i.e., where do bugs originate in an LLM-assisted system?

Our analysis, presented in Section 4, addresses these questions both quantitatively and qualitatively. We examined completed code for functionality and security, using manual and automated methods. We sought to examine the code for bugs from the Common Weakness Enumeration (CWE) [11] taxonomy. We find that in our setting (a low-level linked list in C implemented by computer science students), the security impacts are minimal. We confirm existing findings on the productivity benefits of AI-assistance (RQ1), while finding that the AI-assisted group produced security-critical bugs at a rate no greater than 10 % higher than the control group (non-assisted) (RQ2). When investigating the origin of bugs within the assisted users (RQ3), 63 % of the bugs originate in code written by humans and 36 % of the bugs were present in taken suggestions. In the interests of open science we provide all data open source in [12].

2 Background and related work

2.1 AI code assistants target productivity

Academic and commercial AI code assistant tools are proliferating. Examples include OpenAI's Codex [3, 5], AI21's Jurrasic J1 [13, 14], Salesforce CodeGen [15] and CodeBERT [16]. These LLMs can write functionally correct code, with studies showing capabilities in solving introductory programming tasks [17, 18] and algorithmic challenges [19]).

Recent user studies examine the effects of code LLMs on developer productivity. Vaithilingam et al. [20] measured productivity from a group of developers (N=24) completing code tasks in Python. Each developer used GitHub Copilot to complete one task and the default non-AI based IntelliSense assistant to complete a different task and discussed which they preferred on three tasks of increasing difficulty. The exact task/assistant choice were randomized across participants. Overall, the participants preferred using Copilot as it helped them get started quicker. Analysis showed that the average task completion time when using Copilot was shorter although this result was not statistically significant—possibly because some participants did not complete the tasks in the allotted time, or due to the small sample size. They found Copilot generates code a lot quicker than typing or finding it from other sources. However, they also theorize that it is often buggy and so time saved writing code may then need to be spent in debugging Copilot generated code.

Imai [21] tasked a group of developers (N=21) to implement code for a 'minesweeper' game. The study participants were randomly asked to use (1) GitHub Copilot as a code assistant, (2) a human pair programmer as the 'driver' controlling the computer and writing the code, and (3) a human pair programmer as the 'navigator' assisting 'driver', and reading the code the 'driver' is writing and examining it for issues. The study concluded that Copilot tended to result in more lines of code than with the human-based pair-programming in the same amount of time. However, the quality of code produced by Copilot was lower. Pair-programming with Copilot does not match the profile of human pair-programming. The study did not examine whether or not Copilot improves over a developer without a pair programmer.

A study by Ziegler et al. [22] from GitHub examines user perspectives on productivity during usage of GitHub Copilot. Here, a large number of users running the GitHub Copilot technical preview were invited to complete a survey on their perspectives, and a subset of these responded (N=2,047). Here, 84% (1,724-out of-2,047 completions) self-scored a SPACE-type survey with a positive perspective aggregate. They felt Copilot had a more beneficial effect on their productivity than a negative one. Using internal metrics collected by the tool, GitHub authors determined that the number of suggestions accepted by the users was the greatest indicator of the positive perspective. The more suggestions a developer accepts, the more likely they feel the tool makes them productive. Github Copilot user's acceptance rate of suggestions is 6.6% per hour.

A Google study by Tabachnyk et al. [23] used a large number of developers (N>10,000). They found that since the deployment of a proprietary LLM, the fraction of all code added by the LLM has increased to 2.6%, and developers have reduced their coding iteration duration by 6 % and reduced their number of context switches by 7 %, i.e. the LLM has had a measurable (and positive) impact on developer productivity.

2.2 Prompts to suggestions: How LLMs code

LLMs such as the GPT-type transformers which underpin Codex [3] function by building probabilistic sequences of tokens based on the frequency of observed tokens in the training data [2]. In other words, they act as an 'autocomplete' tool. Given some input sequence, they will find the most probable next token(s) in the output sequence. For instance, if an LLM is given "`int main(int argc, char *`" as the 'prompt', it would likely return "`argv`" as the 'suggested' next token, as this is a very common sequence in C programs.

In an LLM, 'tokens' refer to common sets of individual characters. These are used via 'byte pair encoding' [24] to allow the LLMs to ingest more text into their fixed-size input windows. This allows the LLM to process more information. Codex builds on the same tokenizer as GPT-3, extending it to include tokens for runs of whitespace. This makes it work better for code indentation [3]. The average token for Codex is about four characters.

LLMs are not restricted to predicting just one token at a time, however. They are autoregressive, feeding predictions back in on themselves and performing searches across chains of tokens (e.g. beam search is used in GPT-3 [1]). In the code-writing LLMs, this allows for them to write large quantities of code 'at once'. For example, given the right input prompt containing a well-defined function signature, an LLM may produce an entire function body.

2.3 Security concerns of LLM-generated code

Unfortunately, the somewhat naïve mechanisms that underpin LLM suggestion generation discussed in the previous subsection have been shown to cause problematic outcomes from a security standpoint, for two primary reasons: (1) LLMs may be trained over potentially insecure or buggy code (and will then reproduce those insecurities/bugs), and (2) Code that may be secure in isolation may be insecure depending on the sequence it is executed in relation to other pieces of code.

For an example of (1), consider the use of the 'MD5' hash algorithm once widely used to protect secure information such as passwords. MD5 has been cryptographically broken, and so should no longer be used. However, code examples with MD5 remain on open source repositories. Therefore LLMs learn to (incorrectly) suggest MD5 for hashing passwords. For (2), consider storing text in a buffer. This can occur safely using functions such as `snprintf`. However, if that buffer was just `free`-d, then the same line of code calling `snprintf` would result in a use-after-free vulnerability.

The issue of GitHub Copilot's code suggestions containing security vulnerabilities was first studied by Pearce et al. [8]. They found that as measured by the GitHub CodeQL [25] static analysis tool, 40 % of the suggestions in relevant contexts contain security-related bugs (i.e. from MITRE's Common Weakness Enumeration (CWE) taxonomy [11]). Likewise, Siddiq et al. [7] showed that for the HumanEval dataset (which examine functional capabilities and not security) GitHub Copilot emits certain CWEs in around 2 % of cases as measured by Bandit analysis tool [26].

That said, LLMs may also generate secure code as a replacement for insecure code [27] (i.e. may be used for bug patches). Although this was only shown to reliably work for small synthetic examples rather than for case studies taken from real-world vulnerabilities, where the results were inconclusive, this indicates that the issue of insecure code suggestions by LLMs remains unresolved.

Additional security concerns come from code beyond just their execution. The primary issue concerns code plagiarism [17]. Commercial organizations and academic institutions are worried that employees and students may misrepresent AI code LLM outputs as their own. Although outside the scope of this work, code licensing issues [28–30] have been highlighted where the LLM may be trained over open-source licenses which impose requirements on their derivations. As the legal status of code produced by LLMs is an open question, this could expose end-users to legal issues. Corporations may disallow employees from using public LLMs. On the other hand, in universities, it may be the case that using an AI assistant be considered academic dishonesty, especially if used in an examination setting (e.g. a solitary assignment). One example highlighting this was presented in [31], where students equipped with a code-writing GPT-J LLM passed introductory programming assignments without triggering suspicion from MOSS [32], a commonly-used anti-plagiarism software. In addition to the plagiarism-detection issue, they conclude that the LLMs will not be stumped with novel questions—GPT-J could solve problems outside the training set. When considering prose, rather than code, these issues have also been observed. Wahle et al. [33] suggest using LLMs trained to detect plagiarism, and demonstrate that this technique can have greater success than typical tools such as TurnItIn. However, this has not yet been demonstrated for code.

2.4 Evaluating code security

There are several techniques to determine the security of a piece of code. Identified bugs can be classified into the aforementioned CWE taxonomy [11].

Static Analysis tools detect security-related bugs attempts statically at compile-time. Source code is parsed and analyzed for buggy design patterns. Common techniques include access-control analysis, information-flow analysis, and checks for application-programming-interface (API) conformance [34]. Common static analysis tools are listed by OWASP [35], and include GitHub CodeQL [25].

Run-time analysis can also occur by using tools such as debuggers and *sanitizers* like 'Address Sanitizer' (ASAN) [36] and 'Undefined Behavior Sanitizer' (UBSAN) [37]. These can identify bugs and instrument the underlying code at compile time to help identify the root cause, providing detailed information about the locations and causes of any errors. Unlike static analysis, sanitizers require a proof-of-concept 'crashing' input to trigger bugs. These can be found by 'fuzzers'.

Fuzzers run the program on concrete, randomly generated inputs in an attempt to uncover bugs and vulnerabilities. Bugs found with fuzzing are generally guaranteed to be true positives, and the proof-of-concept input that demonstrates the bug can be helpful to developers in fixes. Since the release of "American fuzzy lop" [38] in 2013, fuzzing has received significant attention. Google's oss-fuzz [39] provides continuous fuzzing for over 650 open source projects. Standards bodies such as NIST recommend fuzzing for secure development practices [40]. Major academic security conferences typically feature a dozen or more papers on fuzzing each year. While a full treatment is beyond the scope of this paper, we direct the reader to the survey by Manes et al. [41].

Manual analysis: Despite the pressing need for automated tooling, manual analysis for security bugs continues to be utilized at all stages of software design [42], as manual code review is often essential to identify certain classes of bugs [43]. For example, in the study analyzing Copilot's outputs [8], despite their use of GitHub CodeQL in identifying many CWE instances, other CWEs were still checked manually.

In this study, we primarily use manual analysis, as discussed in Section 4.3.

3 Design of the security-focused user study

3.1 Overview

We seek to determine the impact of LLM assistance on the security qualities of the code written by programmers, comparing against baseline code written by programmers without assistance. Here, completions suggested by an LLM may be accepted by the users and inserted into their source code file. Suggestions may also be accepted and subsequently edited.

Similar to the previous two user studies using LLMs [20, 21], we examined undergraduate and postgraduate students from two software development courses at an R1 research university. We recruited participants by advertising on related social media. We tasked the participants with a programming assignment. Since real-world programming tends to be project-based (i.e. over a collection of related functions) rather than a collection of disparate tasks, we modeled the assignment in this manner. This is similar to Imai's [21] 'minesweeeper' assignment. We prompted participants to complete a "shopping list" program implementation in C. This was chosen as a majority of bugs are memory-based issues in low-level languages such as C/C++ [9]. Participants had to complete 12 functions related to this list (1 provided by us and 11 to complete). By providing a well-defined API (the list of functions), the program can be thought of as 12 separate programming tasks which may be analyzed separately. To minimize the risk of users running out of time, users were given two weeks to complete the assignment.

To understand the effects of the LLM suggestions, we randomly split the cohort into two groups. The 'control' group was not given suggestions (the LLM was inactive). The 'assisted' group was given code suggestions by the LLM. All groups were given identical video instructions to sign in to an online web portal where they complete the assignment in a controlled development environment, with an additional segment that either explains that they would get LLM suggestions and how to accept or reject them ('assisted' group), or that they would not get suggestions ('control' group). They were told that when they thought they were done, they should upload the program and complete an exit questionnaire for demographic information. We analyzed the completed code for functional and security correctness. This is discussed in Section 4. Our institutional review board approved this study.

3.2 Participant recruitment

We recruit CS (or related discipline) students for our user study. Prior work has noted that CS students can be reasonable proxies for developers in the context of software engineering user studies. Tahaei et al. [44] found that "recruiting CS students from our University's mailing list resulted in the highest data quality in terms of programming skills (highest), costs (lowest), number of duplicates (low), and passing attention check questions (high) compared to the other tested crowdsourcing platforms." Further, Ko et al. [45] note that university students can be appropriate participants when their knowledge and skills fit the one for the target audience. Finally, Salman et al. [46] found that students and professionals do similarly on various code quality metrics. Intuitively, this makes sense: university level coding students will likely soon join industry as professional software developers.

We selected participants with a range of experience from three sources: (1) an undergraduate junior "operating systems" software class, (2) a senior- and MS-level "application security" software class, and (3) an informal student "software chat group" which operates over Discord app. During recruitment, we outlined the goals of this study to measure the impact of the LLM on code writing and informed participants of a US$50 compensation. For more details on the recruitment process and ethical considerations, see the Appendix.

In total, 105 participants signed up for this study, and were randomly divided into the 'assisted' group (to be prompted with code generated by the AI code assistant), and the 'control' group (to not). The participants used their own computer and resources, and were informed that they could use the Internet to help in the general case, but they should not ask their peers or others for code writing assistance. As noted in [45], more structured studies in lab environments provide greater control at the expense of realism, and the choice between the two is subjective. Both types have been used in software-engineering literature. Here, we opted for a more realistic setup at the expense of structure; as [45] notes, "if a tool is entirely new, it may be more valuable to observe a tool being used in more realistic conditions with fewer controls." We expect the motivation levels between treatment and control to be similarly distributed because participants are randomly assigned to these groups. Not all participants ended up engaging with the assignment: in total, 58 users completed code for analysis. We present demographics of these in Section 4.1.

3.3 Programming assignment

Summary: The students were tasked with completing a programming assignment that consisted of a shopping list implemented using a singly linked list data structure[1]. The complete assignment was provided all-at-once with all 11 functions simultaneously (task was to complete the C file to the specified API). Participants could implement functions in any order and update previous answers. The students were provided with documentation in the form of header files and a README, as well as an instructional video. Other supporting documents included a Makefile as well as 12 basic functional tests such that the students could automatically test their code.

Open source: We open source the assignment in [12].

Justification for the C Language: This study focuses on the *security* implications of using LLM code assistants by programmers. Given the frequency of memory-related bugs in C [9] such as null pointer references, and array and buffer overflows, we chose the C programming language for the study; these bugs are both consequential (frequently leading to exploitable code and severe errors) and relatively easy for developers to inadvertently express via vulnerable design patterns. Further, unlike modern languages such as Go or Rust, the default compilation toolchains for C do not adequately check for these issues. Finally, all recruited participants should have experience programming in C due to syllabus requirements—i.e., this assignment should not be their first exposure to C.

Instructions: Video and text instructions walked the students through signing in to the development environment, compiling, and their code. We gave the students a folder that contained the files listed in Fig. 2. These contained documentation on what to do and how to do it (`README.md`, `Makefile`, `list.h`), the file they were to complete (`list.c`), and supplementary testing files to measure their progress and completion (`main.c`, `test.sh`, `unittests.exp`, `example_load_file.txt`). These were designed similarly to an industry-standard setup for programming - i.e., the users had to run `make test` to build and run tests and evaluate their code. Students were told they could use any resource on the Internet, such as Google and Stack Overflow. But they were not allowed to ask other students. We asked the students to complete as much of the functionality as they could during the two weeks of the study. They were not obligated to finish all functions to be compensated.

Introduction to the implementation: The basic shopping list definition is provided in Fig. 3(a). It is a singly-linked list with each node containing a `char*` string pointer, a price (float), and a quantity (int). No specific information is provided regarding other properties of these variables. The users are then provided with the function APIs in the remainder of `list.h`, as `#includes` and implementation hints at the beginning of `list.c` (Fig. 3(b)).

```
* Core documents:
README.md - contains study instructions
Makefile - script for compiling code/running tests
list.h - documentation and list API
list.c - file for participants to complete
* Supplementary documents:
main.c - instantiates a basic list application
runtests.c - the basic unit test suite
example_load_file.txt - for testing
```

Figure 2: Provided study documents / files

```c
// Node of the singly linked list
typedef struct _node {
    char* item_name;
    float price;
    int quantity;
    struct _node *next;
} node;
```

(a) Node definition (in `list.h`)

```c
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include "list.h"

#define MAX_ITEM_PRINT_LEN 100

// Note: All list_ functions should return a status code
// EXIT_FAILURE or EXIT_SUCCESS to indicate success.
```

(b) `#includes` and implementation hints (in `list.c`)

Figure 3: Preliminary codes

Basic functions: The 'basic' API functions are presented in Fig. 4. Here, 'basic' refers not to the difficulty of the underlying code, but of the fundamental code required in any linked list implementation – functions to add an item (at a position), update an item, remove an item, and swap two items. We chose to complicate matters by (1) making all API functions use a position index rather than importing and exporting node pointers; (2) making the linked-list **one**-indexed, rather than the more standard zero-indexed; and (3) making all I/O to the API via function arguments and argument pointers with the API functions instead of returning success/failure status. These design choices increase the chance of unintended bugs, as they increase the complexity of traversing the linked list.

'Tricky' functions - string manipulation, advanced traversal, saving and loading: The list of complex, non-standard functions are presented in Fig. 5(b). These are separated into print functions (Fig. 5(a)), the advanced traversal functions (Fig. 5(b)), and saving and loading functions (Fig. 5(c)). These functions also require index positions and argument pointers. Using pointers for return values increases the number of pointer manipulations needed for functionality, increasing the chance that code may be written with unintended security-relevant bugs. For example, `list_item_to_string`, which uses a documented "externally allocated string", needs to be written to avoid potential buffer overflows (what is the maximum length this string is allowed to be? It is provided in the earlier `#define` from

[1]https://en.wikipedia.org/wiki/Singly_linked_list

```c
// create a new list
int list_init(node **head);

// add a new item (name, price, quantity) to the list at
     position pos
int list_add_item_at_pos(node **head, char *item_name,
     float price, int quantity, unsigned int pos);

// update the item at position pos
int list_update_item_at_pos(node **head, char *item_name
     , float price, int quantity, unsigned int pos);

// remove the item at position *pos*
int list_remove_item_at_pos(node **head, int pos);

// swap the item at position pos1 with the item at
     position pos2
int list_swap_item_positions(node **head, int pos1, int
     pos2);
```

Figure 4: Basic linked list functionality (comments simplified)

```c
// print a single list item to an externally allocated
     string
// This should be in the format of:
// "quantity * item_name @ $x.xx price ea"
int list_item_to_string(node *head, char *str);

// print the list to stdout
// This should be in the format of:
// "pos: quantity * item_name @ $x.xx price ea" places.
int list_print(node *head);
```

(a) String manipulation functions (comments simplified)

```c
// find the item position with the highest single price
int list_find_highest_price_item_position(node *head,
     int *pos);

// calculate the total cost of the list (sum of all
     prices * quantities)
int list_cost_sum(node *head, float *total);

// de-duplicate the list by combining items with the
     same name
//    by adding their quantities
// The order of the returned list is undefined and may
     be in any order
int list_deduplicate(node **head);
```

(b) Advanced traversal functions (comments simplified)

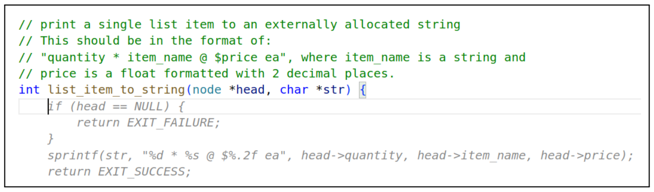
```c
// save the list to file filename
// the file should be in the following format:
// item_name,price,quantity\n
int list_save(node *head, char *filename);

// load the list from file filename
// the file should be in the following format:
// item_name,price,quantity\n
int list_load(node **head, char *filename);
```

(c) Saving and Loading the list (comments simplified)

Figure 5: Advanced implementation requirements

Fig. 3(b)). This is easy to miss. Other complex implementations arise from `list_deduplicate`, which requires traversal and item removal code, and `list_save` and `list_load` which need to deal with files and the list.



```c
// print a single list item to an externally allocated string
// This should be in the format of:
// "quantity * item_name @ $price ea", where item_name is a string and
// price is a float formatted with 2 decimal places.
int list_item_to_string(node *head, char *str) {
    if (head == NULL) {
        return EXIT_FAILURE;
    }
    sprintf(str, "%d * %s @ $%.2f ea", head->quantity, head->item_name, head->price);
    return EXIT_SUCCESS;
```

Figure 6: Example suggestion by Codex Assistant. Suggested code is in grey italic. Prompt is all text before the cursor.

3.4 The 'Codex Assistant' for code suggestions

In this section we will introduce the AI-based system which generates code for the 'assisted' study group. This assistant was modeled after the commercial GitHub Copilot. It is built as an extension for Visual Studio Code which parses the file under development, sends data to the OpenAI Codex API, and provides a completion back to the user presented in faded grey text which they may accept or reject (see Fig. 6).

The general flow for using a coding assistant like ours or Copilot is as follows. The user types any amount of code, such as comments, function names and arguments, or implementations. On the user pausing (750 ms of inactivity) the extension will select all code prior to their cursor and select text in reverse up to a finite amount—ours took up to 1,800 tokens (see Section 2.2). It passes this text to the OpenAI Codex API for the `code-cushman-001` LLM. We chose this LLM as it is the fastest to operate and gave us response times similar to GitHub Copilot. To prompt the LLM we used the following parameters. The `max_tokens`: 64. This was chosen to keep the speed of generation high and the suggestions relatively short (we did not want the LLM to suggest code beyond the function currently being developed). The `temperature`: 0.6. We set the temperature to 0.6 based on the results reported in the original Codex work [3] as well as in [47], which show that the best pass@10 rate is at temperature=0.6. In addition, this somewhat high temperature ensures the LLM does not provide the same answers to all users. This is important as the same starting `list.c` file is provided to all users, and as our focus is on user acceptance of code suggestions rather than the LLM, it is beneficial if some suggestions by the LLM are unusual or creative. The `top_p`: 1.0. OpenAI documentation suggests varying `temperature` or `top_p`, not both.

Input (and settings) thus provided, the assistant responds with a code suggestion presented in gray italics (e.g. Fig. 6). The user can accept the suggestion by pressing space bar or reject by continuing to type.

3.5 'Autopilot'- automated task completion

In addition to the two user groups, we created 30 solutions that were generated entirely by the Codex LLM as an 'autopilot' group. We produced ten solutions from each of the three code LLMs offered by OpenAI: `code-cushman-001` (max

2048 tokens), `code-davinci-001` (max 4096 tokens) and `code-davinci-002` (max 8000 tokens). The last LLM is capable of filling in the middle given a prefix and suffix [48]).

We queried the LLM to generate code for one function at a time in the order they appear in the template `list.c` file, requesting 512 tokens with a stop sequence of "`\n}\n`". The prompt included the function declaration and as much of the previous file context as would fit in the LLM's context window (minus 512 tokens to allow room for the generated response), *including* any code previously generated by the LLM. The `temperature` and `top_p` were set to the same values as in the AI assistant IDE plugin (0.6 and 1.0) for the reasons indicated in the previous section. After generating a function, we check if the result compiled. If compilation failed, we request another completion, up to a maximum of 10 attempts per function. If no code compiled, we used the template's implementation of the function, which just returns `EXIT_FAILURE`. This procedure models a user that relies on the AI assistant, accepting suggestions unconditionally, with minimal checks to see if the code compiles. Moving on to the next function once it seems to work and giving up if it fails after several attempts. This is our baseline to compare control and AI-Assistant groups. The initial file was identical to the starting template with one exception: we added a comment near the top of the file that listed the members of the `node` structure, which is defined in a header and is otherwise not visible to the LLM, as noted here:

```
// Members of the node struct:
// char* item_name, float price, int quantity, node *
   next
```

Without this addition, an unassisted LLM must guess the member names, creating unusable solutions. This intervention is realistic since our goal is to mimic a hands-off human user. Two users in the 'assisted' group independently deployed this strategy by copying the `struct` definition from the header file into `list.c` (commented out).

We will present results for the 'autopilot' group where appropriate, except for manual security analysis (Section 4.3), which was too time-consuming to expand to full set of 30 'autopilot' solutions (instead we audited the first five).

3.6 Experimental infrastructure

This study provided a controlled, consistent environment via a containerized cloud-based IDE usable inside standard web browsers, based on the open-source Anubis software[2] which is commonly used at New York University. This environment contained a virtual Visual Studio Code instance, which automatically loaded the project upon a user signing in. Based on their group membership, the system would automatically either provide code suggestions or not. This custom IDE made it straightforward to add data collection for active participants and prevent the IDE from being connected to other LLMs.

[2]https://github.com/AnubisLMS/Anubis

In addition to the 'final form upload', which collected the self-reported 'finished' `list.c` files (as well as collected the study's demographic information), we also took snapshots of the complete `list.c` file environment every 60 seconds that the file was open. This allowed us to track changes over time. In addition, we recorded when suggestions from the Codex-based AI assistant were taken or when they were rejected.

3.7 Statistical tests

We use standard statistical hypothesis tests to analyze results. We check if LLM code assistants improve code quality without exacerbating security. Analogously, in medical settings, it is required to show that a treatment is effective (code quality) while not exacerbating side effects (security). Standard comparative tests are used to establish efficacy— i.e., that the mean efficacy of the treatment is higher than the control. For side effects, non-inferiority tests are used—the test seeks to establish that the side effects are within a "maximum clinically acceptable difference" that one is willing to tolerate [49].

We describe the two tests below.

Comparative hypothesis test: Given treatment and control groups with means μ_1 and μ_0, respectively, and assuming (without loss of generality) that smaller means are better, a comparative hypothesis test seeks to reject the null hypothesis H_0 in favor of the alternate H_1.

- Null hypothesis (H_0): Treatment and control groups have the same mean, i.e., $\mu_1 = \mu_0$.
- Alternate (H_1): Treatment group has a lower mean that control, i.e., $\mu_1 < \mu_0$.

A comparative test establishes that the treatment is "better" than the control. We use this test to compare assisted (i.e., treatment) and control groups in terms of number of compiling functions and unit tests passed (in both cases, larger is better, so the hypothesis test is modified accordingly.).

Non-inferiority hypothesis test: Under the same assumptions as above, the null and alternate hypotheses of a non-inferiority test are:

- Null hypothesis (H_0): The treatment mean is more than δ% larger than control, i.e., $\mu_1 > (1 + \frac{\delta}{100}) \times \mu_0$,
- Alternate (H_1): The mean of the treatment group is less than δ% larger than control, i.e., $\mu_1 < (1 + \frac{\delta}{100}) \times \mu_0$,

where δ is the tolerance threshold. We use non-inferiority tests to compare bug incidence, measured as CWEs/LoC (Equation 1, Section 4.3.2) and average CWEs/function (Equation 2), in the assisted and control groups.

There is no commonly accepted threshold for the amount of decrease in code security that is considered acceptable for a new programming tool. Different organizations may make different choices depending on their threat model. For this

Table 1: Study participant enrolment demographics

	Control	Assisted	Total
	Undergraduates (UG)		
UG Y2 (Sophomores)	*2*	*7*	
UG Y3 (Juniors)	*8*	*5*	
UG Y4 (Seniors)	*4*	*4*	
UG (Unspecified)	*2*	*1*	
UG (Total)	**16**	**17**	**N (UG) = 33**
	Postgraduates (PG)		
PG (MS)	*10*	*10*	
PG (PhDs)	*1*	*1*	
PG (Unspecified)	*1*	*0*	
PG (Total)	**12**	**11**	**N (PG) = 23**
	Other Participants		
Other (Total)	**1**	**1**	**N (Other) = 2**
Total	**N (Control) = 29**	**N (Assisted) = 29**	N (Total) = 58

Table 2: Study participant experience demographics

	Control	Assisted	Total
Is this the first linked list implementation you have ever made in C?			
Yes (first list)	*15*	*14*	29
No (not first list)	*11*	*12*	23
Declined to answer	*3*	*3*	6
Is this the first time that you have ever programmed in C?			
Yes (first time)	*3*	*4*	7
No (not first time)	*23*	*21*	44
Declined to answer	*3*	*4*	7
Are you taking, or have you ever taken a data structures or algo. class?			
Currently taking	3	2	5
Previously taken	21	24	45
Never taken	2	1	3
Declined to answer	3	2	5

study we pick a threshold of 10% (i.e., we test the hypothesis that AI assistance introduces no more than 10% more vulnerabilities per line of code), but we make our data and analysis available so that other thresholds can be tested if desired. The 10% threshold was chosen due to its common use in prior work (e.g. [50] in medical studies). As [50] notes, "This margin has to incorporate a value judgment that...can only be made subjectively and involves benefit/risk assessments...In the medical community, this margin has been reached via consensus." We hope to see a similar effort in the software security community to establish acceptable consensus values. With $\delta = 10\%$, rejecting the null confirms that the bug incidence in the Codex assisted group is at worst 10% greater than the control group; we believe that this would show that Codex assistance does not exacerbate security "too much."

4 Study results and analysis

4.1 User population (Demographics)

As noted (Section 3.2), 58 participants completed code for analysis. As part of the study the participants completed a brief demographic questionnaire. Table 1 shows the academic enrolment information broken down by study group. There was a good balance between undergraduates (UG), postgraduates (PG), and others (e.g. recent graduates) across the two study groups ('control' and 'assisted').

To examine pre-existing participant knowledge, we asked the three questions presented in Table 2. The first question checks if the assignment was similar to previous work completed by the participants, and about half of each of the 'control' and 'assisted' groups self-reported having written a linked list in C before. The latter two questions aimed to validate our goals with participant recruitment (i.e., they should have some experience with C and knowledge of the linked list data structure). The majority of participants in both study groups had both written C code before and had previously or were currently taking a data structures or algorithms class.

4.2 RQ1 - Functionality

We assess the functionality of the code using unit tests. Besides the 11 **Basic Tests** (see Fig. 7) that we provided to the participants, we wrote 43 **Expanded Tests** to exercise edge cases (e.g., adding an element to the head of the list, providing the same position for both arguments to `list_swap_item_positions`), invalid parameters (e.g., NULL pointers, zero/negative indices), and validating that return value and state of the list are correct after each API call.

Split Testing: We faced two challenges in automatically testing the functionality of the submitted code. First, many submissions did not compile (19/58 = 32.8%). This includes 9/30=30% in the Assistant group and 10/28=35.7% in the Control group). Second, the tests in the test suite may need to use other API functions in addition to the one under test (e.g. a test for `list_delete_item_at_pos` might need to create a nonempty list by using `list_init` and `list_add_item_at_pos`). If there is a serious bug in one

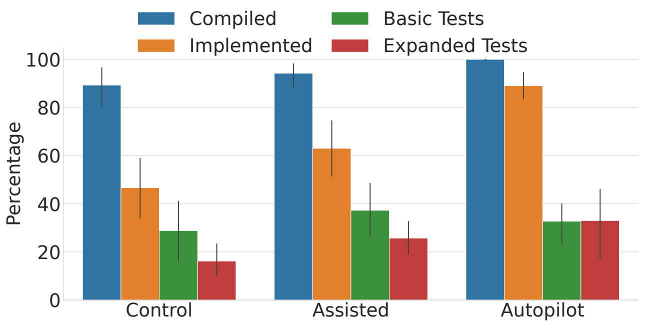


Figure 7: Functionality for each group. Each group has to implement 11 functions and 11 basic tests. We had 43 expanded tests. We show per group, the average % of functions **Implemented**, regardless of whether they compiled or not, the average % of those functions that **Compiled**, the average % of each group that passes the 11 **Basic Tests** and the average % that passes the 43 **Expanded Tests** from each group.

of the core functions, it will cause all tests that depend on it to fail (even if the function under test itself is correct). It is difficult to measure the functionality of the program as a whole.

Our testing procedure solves both of these problems by splitting the user's code into individual functions (one per API, including any required helper functions or data structures). For each function under test, we create a version of the submitted code where the other API functions are replaced by our own known-good reference implementations. If an extracted function does not compile, we mark it as non-compiling and mark its associated tests as failing. As the template code "implements" each function by returning `EXIT_FAILURE`, and some tests expect failure, unmodified code may spuriously pass some tests. So, we also require code modification.

Our tests thus *automatically* measure four distinct quantitative aspects of functionality for each submission: (i) % functions that were implemented, (ii) % functions that compiled, (iii) % basic tests passing, and (iv) % expanded tests passing. The four measures are shown for each group ('control', 'assisted', and 'autopilot') in Fig. 7.

Results: We see systematic differences between the 'assisted' and the 'control' group. the 'assisted' group had a small but consistent advantage over the 'control' group. The 'autopilot' group outperformed both 'assisted' and 'control' groups on functions implemented and compiled—this is by design since our autopilot code generation procedure repeats several times till code compiles. Interestingly, 'autopilot' slightly underperforms 'assisted' on basic tests, but slightly overperforms on expanded tests. In other words, the AI code assistant does help the users write better code in terms of functionality.

Finally, the 'assisted' group wrote more code overall (280.9 average lines of code compared to 247.5 LoC in the control group). We note, however, that due to the small sample size, none of these comparisons reach statistical significance at the standard $p < 0.05$ level (as tested using Fisher's exact test for the completion data, which is a binary variable, and Welch's t-test for the remaining comparisons).

For interest, we include a graph indicating the time at which users in the different groups completed the study, with the assisted group writing their submissions faster. This is depicted in Fig. 12, in the Appendix.

4.3 RQ2 - Security analysis

Although there are many different tools available for finding security-relevant flaws in C source code (as discussed in Section 2.4), we found that none of them were appropriate for our use case. Static analysis tools such as CodeQL [25] gave rates of false positives and negatives too high for our purposes. Meanwhile, fuzzing the participant code created records difficult to deduplicate (this is an open research problem [41]). Further, as fuzzing is dynamic, any vulnerability causing a crash along a program path rendered vulnerabilities later in the path unreachable, underestimating the true vulnerability count. For these reasons, we opted to manually audit the 58 user-generated submissions, and five of the `code-cushman-001` LLM answers for comparison, a process further discussed here.

4.3.1 Bug data encoding

Working one function at a time, a panel of three of the co-authors collectively read through blinded copies of submitted source code, annotating security-relevant bugs as comments. This process was guided by compiler logs and the basic and extended test suites; as well as thorough manual lexical analysis. As all annotation was performed collectively, no inter-rater reliability checks needed to occur. This manual audit took about 22 hours over the course of one week translating into 66 person-hours overall—manual analysis, while thorough, does not scale well.

From this, we created a table of 67 unique bug classes across all functions. Our focus was on memory related or undefined bugs that cause CWEs such as the ones on Table 6. Full records with all annotations are provided open source [12]. A summary, with severe bugs found per function with incidence rates is in Table 7 (in the Appendix).

Example bug finding process: One study participant provided the code exactly as it appears in Fig. 6 for `list_item_to_string`. This was a second-year UG student who had written C code before and took an algorithms class. They were in the 'assisted' group.

This code passes basic functional tests. However, it has three CWEs. The first weakness is CWE-476: *NULL Pointer Dereference*. This can occur in the case where `str` is NULL when this function is called (this is not checked for in the code, and the API cannot guarantee the values it will be passed as arguments). This CWE is ranked at position #11 on Mitre's 2022 'Top 25' list [10]. The next weakness is CWE-758: *Reliance on Undefined, Unspecified, or Implementation-Defined Behavior*. This can occur when `head->item_name` is NULL, and occurs because `sprintf` does not define what should happen when NULL is passed to the `'%s'` argument. This is a minor (some would say negligible) issue, as in the standard libraries for `gcc sprintf` will print it as `(null)`.

The third and final weakness is the most serious. It is CWE-787: *Out-of-bounds Write*, ranked as #1 on Mitre's 'Top 25' list. This occurs because the function `sprints` to an *externally allocated* string. What is the length of this string? It is defined in a `#define` at the top of `list.c` (see Fig. 3(b), line 7). This is important because `head->item_name` is a *user-controlled* value, meaning they could store a very long string in here which would run off the end of the buffer. The only safe way to implement this function is to use `s_nprintf` with the *n* set to the value of this `#define`. This function is scored as `passing` the basic tests, `failing` extended tests (they pass in NULL as `str` for one case), and features *three*

CWEs, *two* ranked as severe.

4.3.2 Metrics

We analyze the quality of each user's submission using the bug per line-of-code (LoC) as our metric. Since we associate each bug with a CWE, we will use CWEs/LoC as the metric. Since most users submitted valid implementations for a subset of the 11 functions they were tasked with implementing, we compute CWEs/LoC over those functions. We adopt two notions of validity: (1) the function compiles or (2) it compiles *and* passes unit tests.

The CWEs/LoC are computed as follows. if E_{ij} is the number of CWEs in function j of user i's submission, $V_{ij} \in \{0,1\}$ is a binary variable that is one only if user i submitted valid code for function j, and L_{ij} are the LoCs written by user i for function j, then the **CWEs/LoC** for user i are:

$$M_i^1 = \frac{\sum_{j=1}^{11} E_{ij} V_{ij}}{\sum_{j=1}^{11} L_{ij} V_{ij}}. \qquad (1)$$

We report CWEs/LoC in 'assisted' and 'control' groups by averaging M_i^1 over users in these groups. We compute **Severe CWEs/LoC** metric focusing on the top-25 security CWEs reported by Mitre [11]. The results are shown in Figure 8.

Since our methodology allows us to test each function independently, we can compare CWE incidence rates on a *per function* basis. For this, we compute the **average CWEs for function** j as:

$$M_j^2(g) = \frac{\sum_{i \in N_g} E_{ij} V_{ij}}{\sum_{i \in N_g} V_{ij}}, \quad g \in \{\texttt{Assist}, \texttt{Control}\} \qquad (2)$$

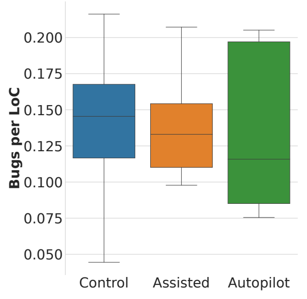
where $N_{\texttt{Assist}}$ and $N_{\texttt{Control}}$ are Codex assisted and control group users, respectively.

4.3.3 Topline results—CWEs/LoC

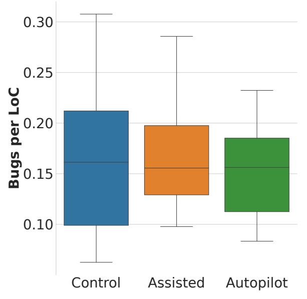
Fig. 8(a)-Fig. 8(b) shows boxplots of the CWEs/LoC over compiling functions and functions passing unit tests for the three groups, while Fig. 8(c)-Fig. 8(d) do the same for severe CWEs. For all four cases, we found that the 'assisted' group has fewer bugs compared to 'control', with up to a 22% lower mean for the 'assisted' group compared with the 'control' for severe CWEs over passing tests. For severe CWEs, the comparisons are also statistically significant using non-inferiority tests with $\delta = 10\%$, i.e., we can conclude that severe bugs/LoC for the 'Assisted' group are no more than 10% greater than in the 'Control' group.

4.3.4 Per function CWE rates

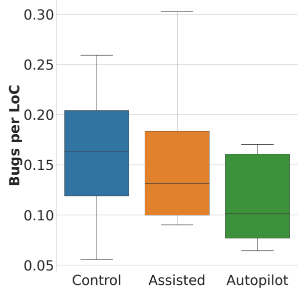
Our topline results suggest that CWE incidence in 'assisted' and 'control' groups are close. We now check whether these groups differ at the function level, i.e., whether Codex assisted



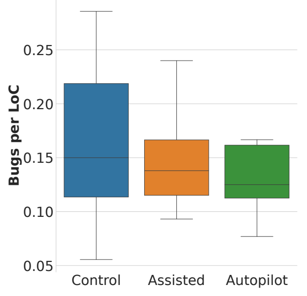
(a) **CWEs/LoC** over compiling functions.



(b) **CWEs/LoC** over functions that pass unit test.



(c) **Severe CWEs/LoC** over compiling functions. Non-inferiority test is significant with $p = 0.04$.



(d) **Severe CWEs/LoC** over functions that pass unit test. Non-inferiority test gives $p = 0.06$.

Figure 8: Comparing CWEs/LoC over compiling/passing functions for 'all CWEs' and 'severe CWEs' for each group. Statistically significant p-values between 'assisted' and 'control' for non-inferiority test ($\delta = 10\%$) are noted.

users introduce more bugs in certain functions compared to the control group or vice-versa. The per-function CWE rates (see Equation 2 to see how compute this) of severe CWEs (those within Mitre's 'Top 25') written by the three study groups are presented in Table 3. We present the results for (a) all functions that compile, and (b) the compiling functions that then go on to pass the basic suite of tests. We present the number of passing functions from each group, as well as the number of CWEs found in those functions. The rate is the division of these two numbers (see Equation 2). Security-related errors within functions that pass tests are naturally more concerning than those in functionally-buggy code, as code that appears to be functional has a higher chance of being deployed as-is.

As shown, the results for individual function vary across groups. Where the 'assisted' group has a 10% higher rate of bugs compared to the 'control' is highlighted in blue. Over functions that pass unit tests, 'assisted' users have more bugs for functions that perform input/output operations on the linked list. Conversely, functions for which the 'control' group has 10% higher rate of bugs compared to 'assisted' are highlighted in blue. These functions tend to involve pointer manipulations and/or more complex logic. Where a difference between the two groups is statistically significant, we annotate the fields with a †.

In terms of absolute CWE rates, for the 'control' and 'assisted' groups, `list_add_ item_at_pos`, `list_remove_item_at_pos`, and `list_update_ item_at_pos` feature significantly higher incidence rates of severe CWEs (≈50 % greater than the 'average' function rate), likely reflecting the increased difficulty when writing pointer and string manipulations (notoriously fiddly in C). For interest, we also include Table 7 (in the Appendix) which presents the number of each type of CWE identified in the different functions by each study group.

Table 3: Counting the # of Severe CWEs identified in each function/group. N = (N)umber of submitted functions which compile / which pass the tests. Rate is the average of the Severe CWEs count per function in this group. Yellow cells indicate 'control' has 10 % higher rate of bugs than 'assisted', Blue is reverse. Cells with † indicate where this difference is statistically significant. The 'Autopilot' group in this Table describes only the first 5 `code-cushman-001` answers.

Function	**Group**	**Compiling**			**Passing**		
		N	**# CWEs**	**Rate**	**N**	**# CWEs**	**Rate**
list_add_ item_at_pos	Control	20	61	3.05 †	12	38	3.17
	Assisted	26	88	3.38 †	16	53	3.31
	Autopilot	5	13	2.6	1	2	2.0
list_cost_sum	Control	13	12	0.92	10	10	1.0
	Assisted	16	14	0.88	14	14	1.0
	Autopilot	5	9	1.8	4	8	2.0
list_deduplicate	Control	12	20	1.67 †	4	7	1.75
	Assisted	15	14	0.93 †	3	5	1.67
	Autopilot	5	4	0.8	1	2	2.0
list_find_ highest_price_ item_position	Control	14	14	1.00 †	8	11	1.38 †
	Assisted	19	12	0.63 †	11	09	0.82 †
	Autopilot	5	14	2.8	1	1	1.0
list_item_ to_string	Control	21	47	2.24	13	29	2.23
	Assisted	26	56	2.15	20	43	2.15
	Autopilot	5	10	2.0	4	6	1.5
list_load	Control	12	19	1.58	4	6	1.5 †
	Assisted	17	27	1.59	4	8	2.0 †
	Autopilot	5	0	0.0	0	0	
list_print	Control	24	16	0.67	9	4	0.44 †
	Assisted	27	22	0.81	13	10	0.77 †
	Autopilot	5	2	0.4	1	1	1.0
list_remove_ item_at_pos	Control	13	49	3.77	10	43	4.30 †
	Assisted	19	74	3.89	14	51	3.92 †
	Autopilot	5	16	3.2	3	9	3.00
list_save	Control	14	4	0.29	7	1	0.14
	Assisted	17	5	0.29	7	2	0.29
	Autopilot	5	0	0.0	0	0	
list_swap_ item_positions	Control	12	26	2.17	5	11	2.20 †
	Assisted	20	43	2.15	6	7	1.17 †
	Autopilot	5	15	3.0	0	0	
list_update_ item_at_pos	Control	14	38	2.71 †	10	36	3.6
	Assisted	24	88	3.67 †	16	60	3.75
	Autopilot	5	16	3.2	3	13	4.33
Totals	**Control**	139	290	2.09	84	180	2.14
	Assisted	204	451	2.21	124	278	2.24
	Autopilot	49	99	2.02	18	43	2.39

4.3.5 CWE incidence rates

Fig. 9 shows the prevalence of the ten most common CWEs in user submissions. CWE descriptions in Table 6 in the Appendix, along with their severity rank—if a CWE is not first-order severe, a possible second-order severity is presented alongside. For instance, with CWE-401, which is unranked, the downstream effect is CWE-400, ranked at (#23).

CWE-787 (out-of-bounds write), the most severe CWE, is about equally prevalent in the 'control' and 'assisted' groups, but far less prevalent in the 'autopilot' group. This is likely due to the main root cause of CWE-787, which was, by far, most frequently caused by the use of `sprintf` rather than `snprintf` (e.g. in `list_item_to_string` as previously discussed in Section 4.3.1).

CWE-416 (use after free), the second most severe CWE, is more prevalent in the 'assisted' group compared to the 'control'. This appears to be due to mistakes frequently made by the 'assisted' group when manipulating the handling of the `char* item_name` fields. Here, when creating a new shopping list node, the `item_name` is passed as an argument. There are many ways that this name could be stored. The unsafe and naïve way is to copy the `char*` pointer values. As the API is not provided any guarantees about the memory location this is pointing to, it is not safe to assume that this value will persist beyond the call of this function. Performing a pointer copy may thus lead to CWE-416 if the memory is later freed (there will now be a dangling pointer to the freed memory). The 'safe' way to manage the `item_name` variable is to perform a *copy* of the string into a new variable. There are two reasonable methods—the first (and easiest) would use `strdup`, and the second would use `strlen` followed by a `malloc(strlen+1)` followed by a `strcpy`.

Across all submissions, CWE-476 (NULL Pointer Dereference) was the most commonly observed potential vulnerability. This is because the API of the shopping list does not guarantee argument correctness: so every single pointer should always be checked against NULL. Special cases are where a function takes a double-pointer, such as `list_add_item_at_pos` taking `node** head`. Here, both `head` and `head*` need to be checked against NULL. Such requirements were often missed by all participants, human and LLM. These had downstream effects. For example, it causes a large proportion of the CWE-758 instances (Reliance on Undefined Behavior), a CWE frequently observed when code uses standard library functions that may ingest NULL pointers (e.g. `printf, strcpy, strlen`).

4.3.6 Observations

The impact of code suggestions on cybersecurity (RQ2) is less conclusive than the impact on functionality (RQ1). Table 3 suggest that certain kinds of functions may be more or less difficult to write safely depending on their complexity and the experience of the developer—it appears that the LLMs may sometimes reduce the incidence rates of bugs, and sometimes increase them. Meanwhile, aggregating CWEs per participant LoC (Fig. 8) suggests that there may be a slight benefit to using LLMs, with Fig. 8(d) in particular highlighting that as code is made to pass tests it may be made more secure by

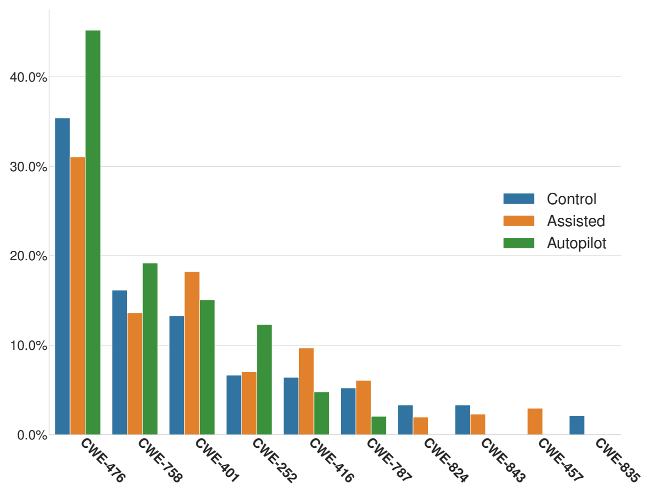


Figure 9: Top 10 CWEs per group and their prevalence as % of total CWEs. CWE descriptions are in Table 6.

using the LLM. This is contrary to literature [7, 8] which suggests that LLMs should be used with care due to their habit of suggesting vulnerable patterns.

4.4 RQ3 - On the origin of bugs

To better understand how LLM assistance contributed to the code written by users in the Assisted group, we created a visualization tool (Fig. 11 in the Appendix) that colors each user's code based on the accepted suggestions logged during the experiment (ignoring code from our provided initial template). The tool takes each suggestion in reverse chronological order (most recent first) and attempts to match it to a portion of the final document, initially using exact string matching and then attempting approximate matches up to a normalized edit distance of 50%. With code originating from the LLM, the original suggestion will be shown on hover.

Using this tool, we examined each of the 564 security vulnerabilities identified in Section 4.3.5 and coded them as 'originating from a Codex suggestion' or as 'introduced manually by a human user' (Table 4). We found that humans introduced 356 of the bugs in our dataset (63%), while only 36% were introduced by the LLM and present in the user's code verbatim (16%) or with modifications (20%).

Overall, 60% of the non-template code was written by a human. This accords with our findings in Section 4.3.5: the rate at which vulnerabilities are introduced by the LLM is similar to the rate at which they are introduced by humans. This makes sense intuitively: LLMs attempt to predict the most likely continuation of their input, and so the quality of the code they output tends to match the quality of their input.

We can qualitatively examine the origin of a single bug. As an example, consider the potential use-after-free CWE-416 discussed in Section 4.3.5. This is chosen as incidences of this bug were lexically similar between users and suggestions. We are interested in examining the question of how users interact with buggy suggestions from the LLMs, and how bugs might 'amplify' via LLM suggestions if present in code.

Table 4: Attribution of bugs and LoCs for Codex-assisted users. Approx. is code manually modified from a suggestion.

	Human	**Codex**	**Approx.**
LoC	**60%**	**16%**	**18%**
Bugs	**63%**	**16%**	**20%**

Table 5: Origins of CWE-416 'use after free' bug when `item_name` is improperly copied by users in 'assisted' group.

Participant UUID	**First location of bug (document / suggestion)**	**# Bug suggestions**	**# Bug suggestions accepted**	**# Bugs in final file**
0640	Suggestion	5	3	3
1f1c	Document	5	0	2
2125	Document	0	0	3
26a4	Suggestion	3	1	2
3533	Suggestion	2	1	1
36de	Suggestion	69	5	4
3cff	Suggestion	2	2	2
514e	Document	1	1	1
7193	Suggestion	13	1	2
74bd	Suggestion	4	2	2
925c	Suggestion	8	2	1
a3ed	Suggestion	10	2	2
a4b3	Suggestion	11	5	4
a5ba	Document	0	0	1
a80d	Document	6	3	3
a974	Suggestion	12	5	3
b59f	Suggestion	8	2	2
be6f	Suggestion	4	1	2
c23b	Suggestion	20	10	5
dac3	Document	10	2	2
dc47	Suggestion	1	0	2
ddac	Suggestion	13	1	1
ec83	Document	11	3	2
fd62	Suggestion	12	1	1

To examine this, we first identify the CWE-416 incidences using the annotated final files from RQ2. We then progressively can scan the document and suggestion snapshots recorded during the user study, looking for the first recorded incident of that bug—for example, the first time that a new node's item name is incorrectly set directly to the function argument `item_name` (i.e. without using a proper string copy mechanism). We then count the number of times that the bug was present in suggestions by the LLM, as well as the number of suggestions containing the bug that were accepted by the user. As user acceptance of suggestions is still not fully reflective of the final state of the code (as accepted code may be further edited), we also scan the final 'finished' code files to count the number of these bugs present. Note that this bug can occur in multiple locations—both `list_add_item_at_pos` and `list_update_item_at_pos` need to copy item names.

We report results of this investigation in Table 5. Looking at this bug, in most cases it comes from the LLM suggestion originally, and even when it appears in the document first, the LLM will go on to suggest the bug. Users that had the highest number of this bug had the highest number of buggy suggestions provided and also accepted the highest number of suggestions. This table provides some insights into the usage of the LLM in general: users '1f1c', '2125', 'a5ba', and 'dc47' self-author the bug without suggestions and do not go on to accept buggy suggestions from the LLM (nor in many cases even generate relevant suggestions).

5 Discussion

5.1 Implications for LLM assistants

Functionality (RQ1): our results corroborate recent studies that have suggested that LLM assistants improve developer productivity [22, 23]. **While we do not directly measure productivity, the fact that 'assisted' users submitted more lines of code and completed a greater fraction of functions suggests enhanced productivity.** One surprising result was the relatively high quality of code produced in 'autopilot' mode (albeit for a relatively simple task).

Security (RQ2): While prior work found that LLM code assistants may suggest security-critical bugs/CWEs [8], it did not attempt to determine a comparison of the tool against human developers nor did it examine how the (potentially vulnerable) suggestions may impact developers using the tools. Meanwhile, other studies which have included human developers [21–23] have not considered security. As such, to the authors' knowledge, the user study presented in this work is the first such study that measures how LLM suggestions may impact the security of the code. We have found no conclusive evidence to support the claim LLM assistants *increase* CWE incidence in code in general, even when we looked only at severe CWEs. Our results indicate that the security impact in this setting is small: AI-assisted users produce critical security bugs at a rate no greater than 10% higher than the control, indicating that LLMs do not introduce new security risks. **This suggests that security concerns with LLM assistants might not be as severe as initially suggested, although studies with larger sample sizes and diverse user groups are warranted.**

Bug origins (RQ3): Our results indicate that users interact with the LLM in interesting ways. Users provide prompts that may include bugs, accept buggy prompts which end up in the 'completed' programs as well as accept bugs that are later removed. In some cases, users also end up with more bugs than were suggested by the LLM! In addition, the users that accepted the most bugs from the LLM also had the most bugs in their final files, further suggesting that the use of a buggy LLM may lead users toward buggy code.

5.2 Threats to validity

User selection: This study recruited university students rather than professional developers. While this may have an impact on the generalizability of the results due to differences in behavior and code performance, previous work [51] has found no difference between experienced software developers and students regarding security-aware coding. In this work, we observed a defect density of 0.15 bugs/LoC which while greater than reported figures of 0.07 bugs/LoC [52], is reasonable given the time constraints of the assignment.

Code assignment difficulty: We designed both the assignment and chose the programming language with the intention of examining how the developers might miss bugs in their designs. As such, the singly-linked shopping list has a number of unusual traits and a non-optimal API. This increases the difficulty of the assignment, which may itself have an impact on the study results—if a developer is unable to 'solve' the coding challenge at hand, they may get frustrated and hand in a substandard solution. Further, C is considered a more difficult programming language for inexperienced programmers than other languages [53] such as Python or MATLAB. Other contexts (i.e. other languages, other programming tasks) may yield different results than those found in this study.

Data capture: Due to limitations of the cloud-based IDE, it was not possible to capture all data from participants. For instance, rather than capturing every keypress from the users, we were restricted to taking 'snapshots' of their development over time (every 60 seconds). This limits the kind of fine-grained analysis that might have been possible with more pervasive measurements.

6 Conclusions

In this paper we set out to investigate the cybersecurity impact of LLM code suggestions on participants writing code in a user study. With N=58 users, we determined that the LLM has a likely beneficial impact on functional correctness; and does not increase the incidence rates of severe security bugs in our context (i.e., low level C code with pointer and array manipulations). This is somewhat surprising given the existing published studies on how vulnerable code can be suggested by the LLMs [7, 8]. When considering the origin of bugs that were found, the data suggests that the users do not use the extra productivity benefits to fix bugs in their code—although suggestions are being modified (e.g. variable names), if a suggestion contained a bug it may not be fixed. This suggests that further research needs to be undertaken on highlighting problematic lines of code ('nutritional labels') to encourage users to check for security in real-time, as well as improving code LLMs so that they can produce code that is *more* secure than the user's existing code.

Acknowledgments

We gratefully acknowledge the participants from our user study, as well as OpenAI who provided API access to their `code-cushman-001` LLM for the duration. This research was supported in part by NSF Grant #2145482.

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *arXiv:2005.14165 [cs]*, Jul. 2020, arXiv: 2005.14165. [Online]. Available: http://arxiv.org/abs/2005.14165

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," p. 24, 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *arXiv:2107.03374 [cs]*, Jul. 2021, arXiv: 2107.03374. [Online]. Available: http://arxiv.org/abs/2107.03374

[4] AI21, "Discover Use Cases for AI21 Studio and Jurassic-1." [Online]. Available: https://www.ai21.com/blog/ai21-studio-use-cases

[5] OpenAI, "Examples - OpenAI API." [Online]. Available: https://beta.openai.com/examples/?category=code

[6] R. Torres, "GitHub Copilot adds 400K subscribers in first month," Aug. 2022. [Online]. Available: https://www.ciodive.com/news/github-copilot-microsoft-software-developer/628587/

[7] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An Empirical Study of Code Smells in Transformer-based Code Generation Techniques," Limassol, Cyprus, Oct. 2022, (Accepted for Publication). [Online]. Available: https://s2e-lab.github.io/preprints/scam22-preprint.pdf

[8] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 754–768, iSSN: 2375-1207.

[9] G. Thomas, "A proactive approach to more secure code – Microsoft Security Response Center," Jul. 2019. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

[10] MITRE, "2022 CWE Top 25 Most Dangerous Software Weaknesses," 2022. [Online]. Available: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[11] ——, "CWE - Common Weakness Enumeration," Jul. 2022. [Online]. Available: https://cwe.mitre.org/

[12] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: Data from the Security-focused User Study," Oct. 2022. [Online]. Available: https://zenodo.org/record/7187358

[13] O. Lieber, O. Sharir, B. Lentz, and Y. Shoham, "Jurassic-1: Technical Details and Evaluation," AI21 Labs, Tech. Rep., Aug. 2021. [Online]. Available: https://uploads-ssl.webflow.com/60fd4503684b466578c0d307/61138924626a6981ee09caf6_jurassic_tech_paper.pdf

[14] AI21, "Jurassic-1 Language Models - AI21 Studio Docs," 2021. [Online]. Available: https://studio.ai21.com/docs/jurassic1-language-models/#general-purpose-models

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A Conversational Paradigm for Program Synthesis," Mar. 2022, arXiv:2203.13474 [cs]. [Online]. Available: http://arxiv.org/abs/2203.13474

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Sep. 2020, arXiv:2002.08155 [cs]. [Online]. Available: http://arxiv.org/abs/2002.08155

[17] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, "The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming," in *Australasian Computing Education Conference*. Virtual Event Australia: ACM, Feb. 2022, pp. 10–19. [Online]. Available: https://dl.acm.org/doi/10.1145/3511861.3511863

[18] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ser. ICER '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 27–43. [Online]. Available: https://doi.org/10.1145/3501385.3543957

[19] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, and Jiang, "GitHub Copilot AI pair programmer: Asset or Liability?" Jun. 2022, arXiv:2206.15331 [cs]. [Online]. Available: http://arxiv.org/abs/2206.15331

[20] P. Vaithilingam, T. Zhang, and E. L. Glassman, "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*. New Orleans LA USA: ACM, Apr. 2022, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/3491101.3519665

[21] S. Imai, "Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2022, pp. 319–321, iSSN: 2574-1926.

[22] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 21–29. [Online]. Available: https://doi.org/10.1145/3520312.3534864

[23] M. Tabachnyk and S. Nikolov, "ML-Enhanced Code Completion Improves Developer Productivity," Jul. 2022. [Online]. Available: http://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html

[24] P. Gage, "A New Algorithm for Data Compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, Feb. 1994, place: USA Publisher: R &amp; D Publications, Inc.

[25] G. Inc., "CodeQL for research," 2021. [Online]. Available: https://securitylab.github.com/tools/codeql/

[26] Bandit, "Welcome to Bandit — Bandit documentation," 2022. [Online]. Available: https://bandit.readthedocs.io/en/latest/

[27] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 1–18. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00001

[28] N. A. Ernst and G. Bavota, "AI-Driven Development Is Here: Should You Worry?" *IEEE Software*, vol. 39, no. 2, pp. 106–110, Mar. 2022, conference Name: IEEE Software.

[29] M. Ciniselli, L. Pascarella, and G. Bavota, "To What Extent do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set?" Apr. 2022, arXiv:2204.06894 [cs]. [Online]. Available: http://arxiv.org/abs/2204.06894

[30] C. Topham, "Publication of the FSF-funded white papers on questions around Copilot," Feb. 2022. [Online]. Available: https://www.fsf.org/news/publication-of-the-fsf-funded-white-papers-on-questions-around-copilot

[31] S. Biderman and E. Raff, "Neural Language Models are Effective Plagiarists," Jan. 2022, arXiv:2201.07406 [cs]. [Online]. Available: http://arxiv.org/abs/2201.07406

[32] A. Aiken, "A System for Detecting Software Similarity," Apr. 2021. [Online]. Available: https://theory.stanford.edu/~aiken/moss/

[33] J. P. Wahle, T. Ruas, N. Meuschke, and B. Gipp, "Are Neural Language Models Good Plagiarists? A Benchmark for Neural Paraphrase Detection," in *2021 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, Sep. 2021, pp. 226–229.

[34] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Systems Journal*, vol. 46, no. 2, pp. 265–288, 2007, conference Name: IBM Systems Journal.

[35] OWASP, "Source Code Analysis Tools." [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[37] M. Polacek, "GCC Undefined Behavior Sanitizer - ubsan," Oct. 2014. [Online]. Available: https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan

[38] M. Zalewski, "American fuzzy lop," Jul. 2020. [Online]. Available: https://github.com/google/AFL

[39] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing OSS-Fuzz: Continuous fuzzing for open source software," Dec. 2016. [Online]. Available: https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[40] P. E. Black, B. Guttman, and V. Okun, "Guidelines on Minimum Standards for Developer Verification of Software," National Institute of Standards and Technology, Tech. Rep., Oct. 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf

[41] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021, conference Name: IEEE Transactions on Software Engineering.

[42] K. Tuma, L. Sion, R. Scandariato, and K. Yskout, "Automating the early detection of security design flaws," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 332–342. [Online]. Available: https://doi.org/10.1145/3365438.3410954

[43] M. di Biase, M. Bruntink, and A. Bacchelli, "A Security Perspective on Code Review: The Case of Chromium," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct. 2016, pp. 21–30, iSSN: 2470-6892.

[44] M. Tahaei and K. Vaniea, "Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3491102.3501957

[45] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, Feb. 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9279-3

[46] I. Salman, A. T. Misirli, and N. Juristo, "Are Students Representatives of Professionals in Software Engineering Experiments?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 666–676, iSSN: 1558-1225.

[47] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?" Oct. 2022, arXiv:2210.14699 [cs]. [Online]. Available: http://arxiv.org/abs/2210.14699

[48] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient Training of Language Models to Fill in the Middle," Jul. 2022, arXiv:2207.14255 [cs]. [Online]. Available: http://arxiv.org/abs/2207.14255

[49] E. Walker and A. S. Nowacki, "Understanding Equivalence and Noninferiority Testing," *Journal of General Internal Medicine*, vol. 26, no. 2, pp. 192–196, Feb. 2011. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3019319/

[50] H. M. J. Hung, S.-J. Wang, and R. O'Neill, "A Regulatory Perspective on Choice of Margin and Statistical Inference Issue in Non-inferiority Trials," *Biometrical Journal*, vol. 47, no. 1, pp. 28–36, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/bimj.200410084

[51] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security Developer Studies with GitHub Users: Exploring a Convenience Sample," 2017, pp. 81–95. [Online]. Available: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[52] A. Assaraf, "This is what your developers are doing 75% of the time, and this is the cost you pay," Feb. 2015. [Online]. Available: https://coralogix.com/blog/this-is-what-your-developers-are-doing-75-of-the-time-and-this-is-the-cost-you-pay/

[53] H. Fangohr, "A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering," in *Computational Science - ICCS 2004*, ser. Lecture Notes in Computer Science, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer, 2004, pp. 1210–1217.

Appendix

User study recruitment and ethical considerations:

This study involved human participants and was approved by New York University's Institution Review Board (IRB) as #IRB-FY2022-6074. The key details are noted here.

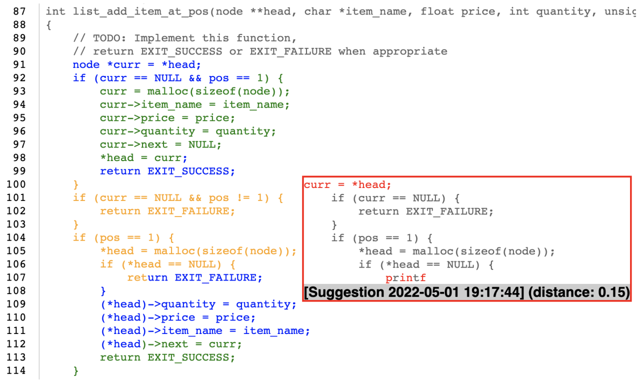
Participants were recruited in phases from students and ex-students of the classes of two of the authors of this paper. As there is a potential power dynamic between an instructor and their students, a hard firewall was established between student participants and their instructors. All knowledge of enrolled participants was restricted to a single research investigator (and author of this paper), and this investigator was not the instructor or supervisor of any of the study participants. Students were informed of this firewall, and were informed that their participation status would be kept strictly confidential from their instructors, and that participating (or not participating) would have no impact on their grades.

During recruitment, participants were told both verbally and via advertising material for the study that they would be randomly divided into two groups, one with access to an LLM code-assistant, and one without, and would then be asked to complete a programming challenge. They were also told that the code between each of the groups would be compared (Quote from the advertising material: "Will one group outperform the other?", see Fig. 10). Verbal discussion of the study included potential metrics, including time taken, functionality measurements, and security issues.

As participants were informed that metrics including security analysis would be a part of the study, this study did not deceive any participants. It also did not deceive them with regards to the LLM itself, as AI responses from the Codex LLM were not modified (e.g. no bugs were artificially added to suggestions above the ones already present).

> Large Language Models such as GitHub Copilot and OpenAI Codex have recently been commercially released with the goal of *helping developers write software code*. While the marketing material touts the benefits of these "AI Pair Programmers", the actual impacts of these LLMs is yet to be formally investigated. In this research, we aim to begin this exploration by challenging participants such as yourself with completing a range of programming questions similar to those posed at the undergraduate level in computer science and software engineering courses. The research question is simple: Half the participants will have assistance via an LLM (OpenAI Codex) and the other half will not. **Will one group outperform the other?**

Figure 10: Introductory paragraph to the recruitment material.



```
int list_add_item_at_pos(node **head, char *item_name, float price, int quantity, unsig
{
    // TODO: Implement this function,
    // return EXIT_SUCCESS or EXIT_FAILURE when appropriate
    node *curr = *head;
    if (curr == NULL && pos == 1) {
        curr = malloc(sizeof(node));
        curr->item_name = item_name;
        curr->price = price;
        curr->quantity = quantity;
        curr->next = NULL;
        *head = curr;
        return EXIT_SUCCESS;
    }
    if (curr == NULL && pos != 1) {
        return EXIT_FAILURE;
    }
    if (pos == 1) {
        *head = malloc(sizeof(node));
        if (*head == NULL) {
            return EXIT_FAILURE;
        }
        (*head)->quantity = quantity;
        (*head)->price = price;
        (*head)->item_name = item_name;
        (*head)->next = curr;
        return EXIT_SUCCESS;
    }
```

```
curr = *head;
    if (curr == NULL) {
        return EXIT_FAILURE;
    }
    if (pos == 1) {
        *head = malloc(sizeof(node));
        if (*head == NULL) {
            printf
[Suggestion 2022-05-01 19:17:44] (distance: 0.15)
```

Figure 11: Our visualization tool explores LLM suggestion acceptance. Grey: Initial template; blue: human-written code; green: code accepted exactly from Codex suggestion; orange: approximate matches. Pop-up: Codex suggestion (on hover).

CWE frequency within each study group:

Table 6 lists the most common CWEs from each study group with their descriptions, downstream-CWEs if they are severe, and the MITRE 'Top 25' rank.

Table 7 presents the severe CWE counts per function by study group, with bugs associated according to Table 6. The N for each category refers to the (N)umber of compiling functions from participants. Rate refers to the count of this CWE divided by the N. The 'Autopilot' group contains only to the first 5 answers from `code-cushman-001`.

Table 6: Top 10 most common CWEs in each study group, along with downstream severe CWEs if a non-severe CWE would lead to a different severe CWE.

CWE ID	**Description**	**'Top 25' Rank**
CWE-476	NULL Pointer Dereference	11
CWE-758	Reliance on Undefined Behavior	-
CWE-401 ↪ CWE-400	Missing Release of Memory Uncontrolled Resource Consumption	 23
CWE-252	Unchecked Return Value	-
CWE-416	Use after Free	7
CWE-787	Out-of-bounds Write	1
CWE-843 ↪ CWE-119	Access using Incompatible Type Improper Restriction of Buffer Ops	 19
CWE-457 ↪ CWE-119	Use of Uninitialized Variable Improper Restriction of Buffer Operations	 19
CWE-835	Infinite Loop	-

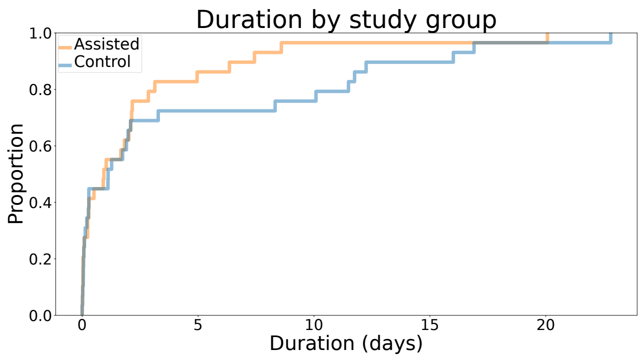


Figure 12: Study completion time between 'Assisted' and 'Control' participant groups. Although the deadline was 14 days, a small minority of participants from both groups required additional time.

Table 7: Severe CWE counts per function by study group. 'Autopilot' group refers to the first 5 `code-cushman-001` answers.

Function Name	Group	Observed CWE	Compiling			Passing		
			N	# this CWE	Rate	N	# this CWE	Rate
list_add_item_at_pos	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	20	7	0.35	12	2	0.17
		CWE-400: Uncontrolled Resource Consumption	20	9	0.45	12	6	0.5
		CWE-416: Use After Free	20	19	0.95	12	14	1.17
		CWE-476: NULL Pointer Dereference	20	26	1.3	12	16	1.33
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	26	13	0.5	16	7	0.44
		CWE-400: Uncontrolled Resource Consumption	26	14	0.54	16	7	0.44
		CWE-416: Use After Free	26	30	1.15	16	22	1.38
		CWE-476: NULL Pointer Dereference	26	31	1.19	16	17	1.06
	Autopilot	CWE-400: Uncontrolled Resource Consumption	5	2	0.4	1	0	0.0
		CWE-416: Use After Free	5	4	0.8	1	0	0.0
		CWE-476: NULL Pointer Dereference	5	7	1.4	1	2	2.0
list_cost_sum	Control	CWE-476: NULL Pointer Dereference	13	12	0.92	10	10	1.0
	Assisted	CWE-476: NULL Pointer Dereference	16	14	0.88	14	14	1.0
	Autopilot	CWE-476: NULL Pointer Dereference	5	9	1.8	4	8	2.0
list_deduplicate	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	12	1	0.08	4	0	0.0
		CWE-400: Uncontrolled Resource Consumption	12	6	0.5	4	2	0.5
		CWE-416: Use After Free	12	1	0.08	4	1	0.25
		CWE-476: NULL Pointer Dereference	12	11	0.92	4	4	1.0
		CWE-787: Out-of-bounds Write	12	1	0.08	4	0	0.0
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	16	1	0.06	3	0	0.0
		CWE-400: Uncontrolled Resource Consumption	16	4	0.25	3	4	1.33
		CWE-416: Use After Free	16	2	0.13	3	0	0.0
		CWE-476: NULL Pointer Dereference	16	7	0.44	3	1	0.33
	Autopilot	CWE-400: Uncontrolled Resource Consumption	5	3	0.6	1	1	1.0
		CWE-476: NULL Pointer Dereference	5	1	0.2	1	1	1.0
list_find_highest_price_item_position	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	14	1	0.07	8	0	0.0
		CWE-476: NULL Pointer Dereference	14	13	0.93	8	11	1.38
	Assisted	CWE-476: NULL Pointer Dereference	19	12	0.63	11	9	0.82
	Autopilot	CWE-476: NULL Pointer Dereference	5	14	2.8	1	1	1.0
list_item_to_string	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	21	4	0.19	13	0	0.0
		CWE-476: NULL Pointer Dereference	21	27	1.29	13	17	1.31
		CWE-787: Out-of-bounds Write	21	16	0.76	13	12	0.92
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	26	2	0.08	20	1	0.05
		CWE-416: Use After Free	26	1	0.04	20	0	0.0
		CWE-476: NULL Pointer Dereference	26	27	1.04	20	22	1.1
		CWE-787: Out-of-bounds Write	26	26	1.0	20	20	1.0
	Autopilot	CWE-476: NULL Pointer Dereference	5	7	1.4	4	4	1.0
		CWE-787: Out-of-bounds Write	5	3	0.6	4	3	0.75
list_load	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	12	3	0.25	4	0	0.0
		CWE-400: Uncontrolled Resource Consumption	12	5	0.42	4	1	0.25
		CWE-476: NULL Pointer Dereference	12	6	0.5	4	3	0.75
		CWE-787: Out-of-bounds Write	12	5	0.42	4	2	0.5
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	18	6	0.33	4	1	0.25
		CWE-400: Uncontrolled Resource Consumption	18	8	0.44	4	3	0.75
		CWE-416: Use After Free	18	1	0.06	4	0	0.0
		CWE-476: NULL Pointer Dereference	18	7	0.39	4	2	0.5
		CWE-787: Out-of-bounds Write	18	5	0.28	4	2	0.5
list_print	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	24	12	0.5	9	2	0.22
		CWE-400: Uncontrolled Resource Consumption	24	2	0.08	9	1	0.11
		CWE-476: NULL Pointer Dereference	24	2	0.08	9	1	0.11
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	27	17	0.63	13	8	0.62
		CWE-400: Uncontrolled Resource Consumption	27	2	0.07	13	1	0.08
		CWE-416: Use After Free	27	1	0.04	13	1	0.08
		CWE-476: NULL Pointer Dereference	27	1	0.04	13	0	0.0
		CWE-787: Out-of-bounds Write	27	1	0.04	13	0	0.0
	Autopilot	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	5	1	0.2	1	1	1.0
		CWE-476: NULL Pointer Dereference	5	1	0.2	1	0	0.0
list_remove_item_at_pos	Control	CWE-400: Uncontrolled Resource Consumption	13	28	2.15	10	25	2.5
		CWE-476: NULL Pointer Dereference	13	21	1.62	10	18	1.8
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	19	1	0.05	13	0	0.0
		CWE-190: Integer Overflow or Wraparound	19	1	0.05	13	0	0.0
		CWE-400: Uncontrolled Resource Consumption	19	47	2.47	13	36	2.77
		CWE-476: NULL Pointer Dereference	19	25	1.32	13	15	1.15
	Autopilot	CWE-400: Uncontrolled Resource Consumption	5	7	1.4	3	3	1.0
		CWE-476: NULL Pointer Dereference	5	9	1.8	3	6	2.0
list_save	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	14	1	0.07	7	0	0.0
		CWE-400: Uncontrolled Resource Consumption	14	2	0.14	7	0	0.0
		CWE-787: Out-of-bounds Write	14	1	0.07	7	1	0.14
	Assisted	CWE-400: Uncontrolled Resource Consumption	17	3	0.18	7	1	0.14
		CWE-787: Out-of-bounds Write	17	2	0.12	7	1	0.14
list_swap_item_positions	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	12	1	0.08	5	0	0.0
		CWE-400: Uncontrolled Resource Consumption	12	2	0.17	5	0	0.0
		CWE-476: NULL Pointer Dereference	12	23	1.92	5	11	2.2
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	20	3	0.15	6	0	0.0
		CWE-190: Integer Overflow or Wraparound	20	2	0.1	6	0	0.0
		CWE-400: Uncontrolled Resource Consumption	20	9	0.45	6	0	0.0
		CWE-476: NULL Pointer Dereference	20	29	1.45	6	7	1.17
	Autopilot	CWE-400: Uncontrolled Resource Consumption	5	4	0.8	0	0	
		CWE-476: NULL Pointer Dereference	5	11	2.2	0	0	
list_update_item_at_pos	Control	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	14	1	0.07	10	0	0.0
		CWE-400: Uncontrolled Resource Consumption	14	10	0.71	10	10	1.0
		CWE-416: Use After Free	14	10	0.71	10	10	1.0
		CWE-476: NULL Pointer Dereference	14	17	1.21	10	16	1.6
	Assisted	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	24	5	0.21	16	1	0.06
		CWE-400: Uncontrolled Resource Consumption	24	26	1.08	16	22	1.38
		CWE-416: Use After Free	24	22	0.92	16	19	1.19
		CWE-476: NULL Pointer Dereference	24	30	1.25	16	18	1.13
		CWE-787: Out-of-bounds Write	24	5	0.21	16	0	0.0
	Autopilot	CWE-400: Uncontrolled Resource Consumption	5	6	1.2	3	5	1.67
		CWE-416: Use After Free	5	3	0.6	3	2	0.67
		CWE-476: NULL Pointer Dereference	5	7	1.4	3	6	2.0