
SOK: EXPLORING HALLUCINATIONS AND SECURITY RISKS IN AI-ASSISTED SOFTWARE DEVELOPMENT WITH INSIGHTS FOR LLM DEPLOYMENT

A PREPRINT

Ariful Haque

Department of Cyber Physical Systems
Clark Atlanta University
Atlanta, GA, USA
mohdariful.haque@students.cau.edu

Sunzida Siddique

Department of Computer Science and Engineering
Dhaka International University
Dhaka, Bangladesh
sunzida15-9667@diu.edu.bd

Md. Mahfuzur Rahman

Silicon Orchard Research and Analytics Lab
Dhaka, Bangladesh
mahim@siliconorchard.com

Ahmed Rafi Hasan

Department of Computer Science and Engineering
Dhaka, Bangladesh
rafihasan@cse.dhaka.edu

Laxmi Rani Das

Department of Computer Science and Engineering
Dhaka, Bangladesh
laxmi.das@cse.dhaka.edu

Marufa Kamal

Department of Computer Science and Engineering
Dhaka, Bangladesh
marufa.kamal@cse.dhaka.edu

Tasnim Masura

Department of Cyber Physical Systems
Clark Atlanta University
Atlanta, GA, USA
tmasura@students.cau.edu

Kishor Datta Gupta

Department of Cyber Physical Systems
Clark Atlanta University
Atlanta, GA, USA
kgupta@cau.edu

ABSTRACT

The integration of Large Language Models (LLMs) such as GitHub Copilot, ChatGPT, Cursor AI, and Codeium AI into software development has revolutionized the coding landscape, offering significant productivity gains, automation, and enhanced debugging capabilities. These tools have proven invaluable for generating code snippets, refactoring existing code, and providing real-time support to developers. However, their widespread adoption also presents notable challenges, particularly in terms of security vulnerabilities, code quality, and ethical concerns. This paper provides a comprehensive analysis of the benefits and risks associated with AI-powered coding tools, drawing on user feedback, security analyses, and practical use cases. We explore the potential for these tools to replicate insecure coding practices, introduce biases, and generate incorrect or non-sensical code (hallucinations). In addition, we discuss the risks of data leaks, intellectual property violations and the need for robust security measures to mitigate these threats. By comparing the features and performance of these tools, we aim to guide developers in making informed decisions about their use, ensuring that the benefits of AI-assisted coding are maximized while minimizing associated risks.

1 Introduction

Large language models have transformed coding by enabling automatic code generation from natural language descriptions. Tools such as ChatGPT, Codium, Copilot, and Cursor AI assist developers in writing, completing,

refactoring, and optimizing code to enhance productivity and improve performance [56]. However, LLMs also pose security risks, as they are trained on publicly available code, which may contain insecure practices [32]. This can lead to vulnerabilities in the generated code that malicious actors could exploit [33]. Furthermore, supply chain attacks targeting third-party services, particularly popular libraries such as npm packages, have increased dramatically in recent years [54]. The use of advanced AI models in software development may unintentionally support these attacks by generating phishing messages and attack plans [11]. Despite their advantages, LLMs can also perpetuate biases and introduce bugs, which can affect performance and raise ethical concerns [44], [19]. LLMs offer advantages in various fields, including healthcare, education, programming, improving communication, workflows, and knowledge discovery [51], [55]. Although LLMs cannot replace human programmers, they streamline tasks such as code writing, documentation, and bug detection, making them increasingly valuable in programming [41]. However, the widespread use of LLMs raises concerns about possible security vulnerabilities and hallucinations [25]. This paper explores these trade-offs with the guidance of developers in leveraging AI's benefits while mitigating associated risks.

1.1 Related work

Heibel et al. [23] discuss Malicious Programming Prompt (MaPP) attacks, where attackers manipulate prompts to make large language models (LLMs) generate vulnerable code. Despite improvements in model capabilities, these attacks remain effective across various platforms. The authors tested MaPP on seven LLMs using HumanEval and Common Weakness Enumerations (CWEs) and found that simple prompts induced vulnerabilities without affecting code correctness. In addition, the best-performing models were more prone to malicious instructions. Their findings suggest that enhancing models alone is not enough to prevent prompt manipulation attacks.

Similarly another author Derner et al. [18] analyzes the security risks of large language models (LLMs) like ChatGPT, focusing on vulnerabilities in content filters and the potential for malicious use. This paper identifies risks such as harmful text generation, data leaks, and unethical content creation. It also highlights the importance of informing policymakers and industry professionals about these issues. The authors evaluate the effectiveness of ChatGPT's content filters and explore the ethical concerns surrounding its use. They conclude that current safeguards are insufficient, allowing for harmful outputs. The study recommends further research to strengthen safeguards and investigate the broader societal effects of LLMs.

In another paper, Oviedo et al. [39] discuss the advanced AI language model ChatGPT, which is widely used for tasks like customer service and chatbots. However, it has drawbacks, including the potential to provide incorrect or unsafe information that may impact user safety. This paper also evaluates ChatGPT's overall effectiveness in promoting safety in contexts such as phone usage while driving and stress management in the workplace.

Alternatively, Jacobi et al. [24] emphasize that the increasing complexity and frequency of cyber threats necessitate updated approaches to strengthen Governance, Risk, and Compliance (GRC) frameworks. One promising approach involves the use of artificial intelligence, particularly LLMs like ChatGPT and Google Gemini, to enhance cybersecurity guidance. Research indicates that ChatGPT generally provides more relevant, accurate, and context-appropriate advice compared to Google Gemini. While both models have limitations, this study highlights the potential benefits of integrating LLMs into GRC frameworks, especially when combined with human expertise to address complex issues.

In another case, Atzori et al. [9] examine on LLMs such as ChatGPT, GPT-4, Claude, and Bard can be misused to create phishing attacks. These models can generate realistic phishing emails and websites without modification, allowing attackers to easily scale their efforts using malicious prompts. To counter this threat, researchers have developed a BERT-based detection tool that effectively identifies phishing prompts across various LLMs. BERT-based detection tool that effectively identifies phishing prompts across various LLMs.

Another author, Latif et al. [45] discuss previous research in code generation using Large Language Models (LLMs), which usually focus on functional correctness but overlook security. It introduces the SALLM framework, which adds security-specific prompts and metrics to evaluate secure code generation. While earlier studies mainly checked syntactic and functional accuracy. This paper improves on that by using a rule-based repair system to enhance syntactic correctness, resulting in better compilation rates, especially for GPT-4. However, the study acknowledges challenges like potential bias in manually created prompts and the lack of real-world task representation, suggesting that future research should improve datasets and security evaluation methods.

Mishra et al. [31] introduced FAVA-Bench dataset for fine-grained hallucination detection in language models (LMs), featuring a comprehensive taxonomy of hallucination types. The dataset includes span-level human annotations and responses from models like Llama2-Chat and ChatGPT. It comes from a number of different sources, such as the No Robots dataset, the WebNLG dataset, and Open Assistant queries. They developed FAVA, a retrieval-augmented LM fine-tuned for detecting and mitigating hallucinations, which significantly outperformed existing systems like GPT-4. The study highlighted that over 60% of errors in model outputs were unverifiable, emphasizing the prevalence

of hallucinations. While the work provides a robust framework, its focus on specific models limits generalizability, prompting future research to adapt FAVA for broader fact-checking tasks and evaluate it across diverse LM architectures.

1.2 Our Contribution

Key Contributions of our SoK paper are:

- User-Centric Insights:** Incorporates feedback from IT professionals to evaluate the impact of AI tools on productivity, error reduction, and collaboration while addressing issues like hallucinations and contextual errors. (in Sections 2-3).

Evaluation of AI Coding Tools: Provides a comprehensive analysis of tools like GitHub Copilot, ChatGPT, Cursor AI, and Codeium AI, detailing their capabilities, user benefits, and limitations in improving software development workflows. (in Section 3)

Security and Risk Analysis: Identifies critical vulnerabilities, including data leaks, adversarial attacks, and replication of insecure coding practices, and proposes strategies to mitigate these risks. (in sections 4-5)

2 User Feedback Data

This dataset was collected from a survey to gather feedback on AI coding tools. We use Copilot AI, Codium AI, Cursor AI, and ChatGPT as tools. The survey aimed to understand user experience, satisfaction, and areas for improvement. The data set includes responses from 66 individuals. Respondents are collected from a renowned IT company, representing various departments and teams, including the Machine Learning/AI Team, Web Development Team, Mobile Development Team, Software Quality Assurance (QA) Team, Human Resources (HR) & Administration, and Marketing. In our analysis, we evaluate the tool features based on feedback to highlight their strengths and areas for improvement 1.

Features	Cursor AI Rating	Codeium AI Rating	ChatGPT Rating	Copilot Rating
Code Generation	3.70	3.24	4.03	4.14
Code Refactoring	3.59	3.30	3.90	4.0
Code Debugging	3.66	3.19	3.90	4.0
Code Explanation	3.77	3.30	4.20	4.14
Code AutoComplete	3.88	3.38	N/A	4.29

Table 1: Comparison of AI Tool Ratings Across Key Features

Analysis shows **ChatGPT** excelled in features of *Code Generation*, *Code Refactoring*, and *Code Explanation*, making it a versatile and highly capable tool for developers seeking comprehensive support. **Cursor AI** demonstrated balanced performance across all categories, leading specifically in *Code AutoComplete* with a high rating of **3.88**. **Codeium AI**, while consistent, scored the lowest overall, indicating areas where improvements are needed to compete effectively with other tools. **Copilot** delivered strong results across the board, particularly in *Code Explanation* (**4.00**) and *Code AutoComplete* (**3.92**), demonstrating its ability as a multifaceted developer assistance provider. Below is a summary of the key insights and feedback from user responses regarding their experiences with various AI coding tools.

We have analyzed the feedback for each tool to determine which tool provided the best responses. The sentiment counts for each ChatGPT, Codeium, Copilot, and Cursor AI model are categorized as positive, negative, or neutral. Below is the sentiment distribution based on user feedback:

Based on the feedback the analysis found that ChatGPT received the highest positive sentiment, with 46 positive ratings, 8 negative ratings, and 12 neutral ratings. Users provide feedback on ChatGPT for its accuracy and natural language processing capabilities, making it a powerful tool for various tasks. However, some users pointed out occasional issues with context understanding in complex scenarios. Conversely, Codeium AI followed with 38 positive responses, 13 negative responses, and 15 neutral responses. Users appreciated Codeium's coding support and speed, but some users found its responses lacking in-depth documentation. On the other hand, Copilot received a more mixed response, with 27 positive, 25 negative, and 14 neutral ratings. While it was valued for its integration with development environments and helpful suggestions, users observed that it occasionally produced irrelevant or incomplete code. Lastly, Cursor AI provides 40 positive responses, 16 negative responses, and 10 neutral responses. Its ease of use and ability to handle specific tasks such as design and content generation, although some users expressed concerns about occasional misinterpretation of input. Overall, ChatGPT received the highest ratings among AI tools, while Copilot experienced a more balanced response with a significant proportion of negative feedback. Below a figure is 3 illustrated for overall sentiment distribution, with "Positive" highlighted for better visibility.

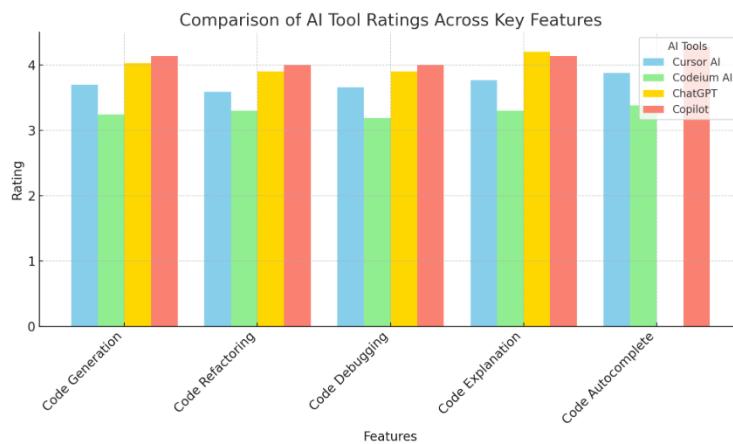


Figure 1: Comparing the ratings of different AI tools

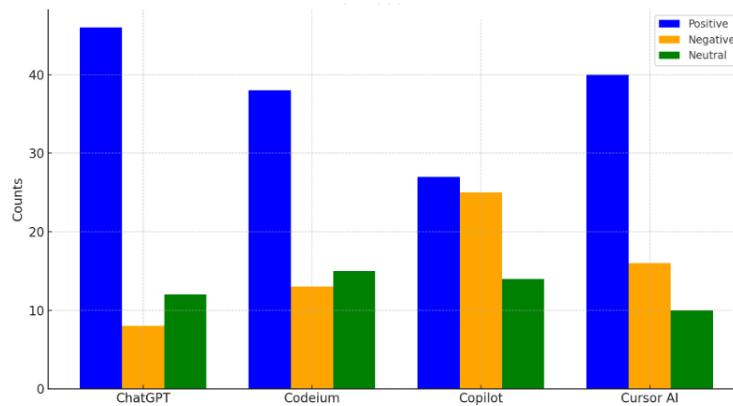


Figure 2: Each satisfaction Sentiment Counts

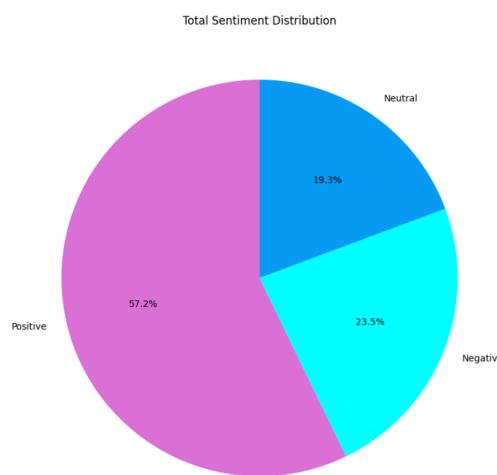


Figure 3: Each satisfaction Sentiment Counts

These AI tools (ChatGPT, Codeium, Copilot, and Cursor AI) are making a significant impact by enhancing productivity, efficiency, and creativity across various domains. Analyzing sentiment data reveals a remarkable trend: the majority of

feedback highlights positive experiences with these tools. This figure 3 reveals that AI tools largely contribute to user satisfaction.

3 LLM tools for Code Generation:

This section explores ChatGPT, an advanced conversational AI model by OpenAI used for generating human-like text and assisting with diverse tasks. It also examines Codium Copilot, a coding assistant designed to boost productivity by providing intelligent code suggestions and debugging support. Additionally, Cursor AI is discussed as a tool that enhances developer workflows through contextual assistance. These technologies highlight the growing integration of AI in improving efficiency and user experiences across various domains.

3.1 Copilot

GitHub Copilot is an AI-powered coding assistant that helps with software development by making code suggestions and giving advice in real time and across situations. It automates common coding tasks so that devs can work together to solve problems.

3.1.1 Evaluation

Initially, GitHub served as an inline coding assistant that offered code suggestions based on the context of previous code. It primarily aids with boilerplate code, but now includes an in-editor chatbot that enables users to input prompts and codebases, receiving customized responses to their queries [9][31][19]. During an in-depth evaluation of GitHub Copilot on a computer vision project, several issues were observed, particularly during the debugging process [10][51][25]. Despite leveraging the GPT-4o model, Copilot's responses were often inadequate, with repeated hallucinations and redundant suggestions that failed to resolve the underlying issues[22]. During the process of using the capability that enables Copilot to combine codebases and files into queries, this issue became more noticeable[50][17][8].

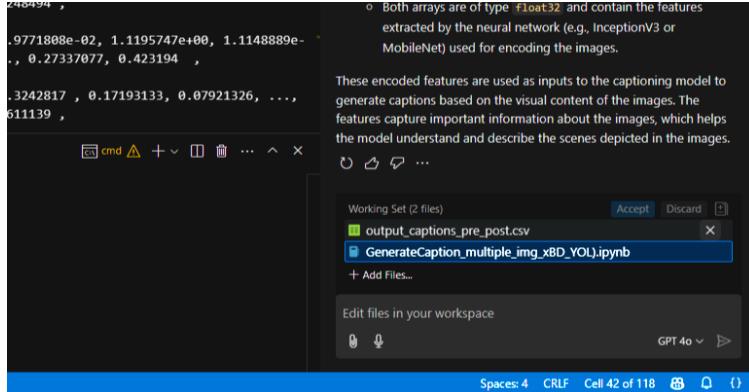


Figure 4: Files and Codebase integrating

Copilot provides the capability to interact with the codebase of a project or link notebook files, the majority of the responses it provides are textual explanations, with suggested code snippets to be added in for good measure. It is important to note that the practical applicability of these proposals is restricted because they do not provide specific guidance on where or how to implement them[15]. Furthermore, the functionality that allowed users to directly apply suggested modifications (via the three-dot menu next to the response) frequently provided results that were ineffective in terms of resolving the issue. These observations highlight challenges in effectively utilizing Copilot's features in complex debugging scenarios.

3.1.2 User Feedback Overview

During our user survey, it was found that Copilot significantly reduces code review and refactoring time by 15 to 30 minutes per task. Furthermore, users rated the quality of the Copilot generated code as follows: 14.3% at 60%, 57.1% at 70%, 14.3% at 80%, and 4.3% at 85%. Its error identification effectiveness received an average rating of 3.85 out of 5.

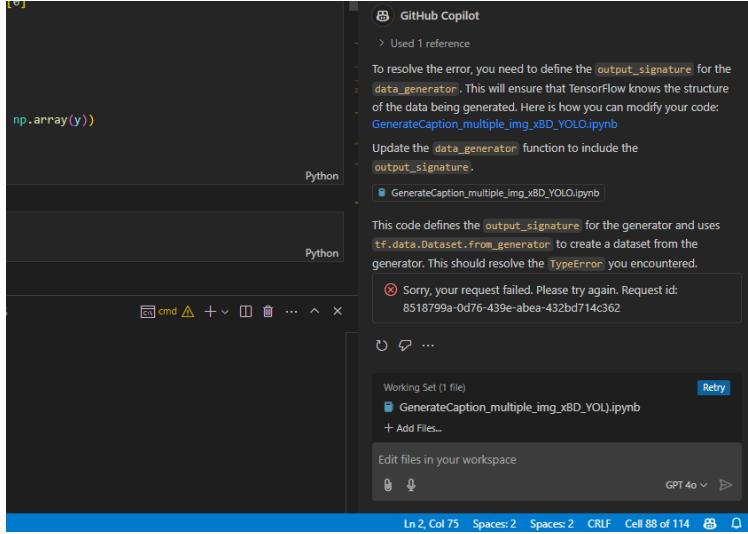


Figure 5: Github Copilot unable to grasp user request

3.1.3 Security Analysis

GitHub Copilot poses security risks due to its dependence on data sets and integration systems. The Enterprise version allows training on private repositories, which may result in unintentional data leaks if improperly handled, even if it is primarily trained on public repositories to minimize the exposure of sensitive data. Additional dangers may arise from sending sensitive requests to third-party services when using external integrations, such as Bing Search, for information gathering. Additionally, Copilot might duplicate vulnerabilities from its training data, advise out-of-date or unsafe dependencies, or conjure up nonexistent package names, all of which could lead to exploitation chances. Because Copilot can produce code fragments that look like copyrighted content without giving proper credit, concerns around intellectual property also surface. Best practices like code review, automated security tool use, and adherence to safe coding principles are advised in order to reduce these risks.

Recent studies have highlighted potential security vulnerabilities in Microsoft’s Co-Pilot could inadvertently expose confidential information, such as passwords and API keys, from its training data [22]. Also, Co-Pilot Studio has a major security hole (CVE-2024-38206) that could be used to steal sensitive information using a server-side request forgery (SSRF) attack [50]. During the Black Hat USA 2024 conference, experts pointed out several security weaknesses in Co-Pilot that might permit unauthorized access to confidential data and corporate credentials [17]. One of the major issues involved in LLM is adversarial attacks, which occur when inputs are manipulated to generate inaccurate output. Even minor modifications to the input data can result in incorrect answers[16]. Another challenge is data poisoning, where harmful data is deliberately inserted into the model’s training set, causing biased or inaccurate results[16]. Furthermore, there is a security threat from prompt injection, which allows attackers to alter the prompts given to LLMs and retrieve sensitive information[16]. Training can also introduce covert vulnerabilities or backdoors, allowing attackers to obtain unauthorized access to systems or controls [16].

3.1.4 Case study

It is possible to use GitHub Copilot for fundamental analyses by a coding assistant, such as the generation of code, the explanation of code, and the completion of code. Copilot is able to leverage open-source GPT models; it can also be utilized for mistake correction and the debugging process. For example, in a python file, create a method by typing the method name. GitHub Copilot will automatically suggest a method body in grayed text. To accept the suggestion, press Tab. As an illustration of visualization, we will now give the use cases of Copilot which are accompanied by appropriate figures:

A. Code Suggestions and Generation Code completions are offered by GitHub Copilot, which also converts natural language prompts into coding recommendations specific to the context and style of a project. It makes use of a machine learning technique built on the Generative Pre-trained Transformer (GPT) model from OpenAI, which has been extensively trained on open-source code. To produce pertinent recommendations, this deep neural network analyses the context of the code.

The screenshot shows a code editor window in Visual Studio Code. At the top, there's a toolbar with icons for 'Generate', 'Code', 'Markdown', 'Run All', 'Restart', and 'Clear All Output'. Below the toolbar, the code editor displays Python code. A blue vertical bar on the left indicates where code suggestions are being proposed. The code includes imports for numpy, torch, and transformers, followed by a function definition for reading text from a file.

```

home > mahim > Workspace > ml_glm > GLLM_exp.ipynb > import faiss
Generate + Code + Markdown | Run All Restart Clear All Output
import faiss
import numpy as np
import torch
from transformers import AutoTokenizer, AutoModel

# Function to read text data from a file
def read_text_file(file_path):
    with open(file_path, "r") as file:
        data = file.read()
    return data

```

Figure 6: Code suggestions proposed by copilot

To generate code from a prompt utilizing GitHub Copilot, the user must input a natural language description of the desired code functionality in the Copilot chat window within their IDE (such as Visual Studio Code).

The screenshot shows a GitHub Copilot interface within a code editor. The user has typed 'I want to build a RAG system. Help me implement the initial imports.' The Copilot interface suggests a series of imports for a Retrieval-Augmented Generation (RAG) system. The imports listed include transformers, RagTokenizer, RagRetriever, RagSequenceForGeneration, numpy, pandas, json, requests, string, os, multiprocessing, logging, and time.

```

I want to build a RAG system. Help me implement the initial imports.
Untitled-1.ipynb • Cell 1:1-33
GitHub Copilot
Sure, here are the initial imports you might need for a Retrieval-Augmented Generation (RAG) system using Python. This example assumes you are using libraries like `transformers` for the language model and `Faiss` for the retrieval part.

# Import necessary libraries
# For language model
from transformers import RagTokenizer, RagRetriever, RagSequenceForGeneration
# For retrieval
import Faiss
# For encoding
import numpy as np
import pandas as pd
# For text processing
import string
# For handling JSON data
import json
# For handling HTTP requests (if needed)
import requests
# For logging
import logging
# For system operations
import os
# For parallel processing
from multiprocessing import Pool
# For timing operations
import time

```

Figure 7: Code suggestions proposed by Copilot

Copilot will then propose code snippets based on the user's prompt; the user can subsequently select the most appropriate suggestion and integrate it into their project. Mostly, they can produce code through quick engineering.

The screenshot shows a GitHub Copilot interface within a code editor. The user has asked 'What should I implement next?'. The Copilot interface provides a step-by-step plan for setting up the RAG system: 1. Initialize the Tokenizer, 2. Initialize the Retriever, and 3. Initialize the RAG Model. It then provides the code to implement these steps, which includes initializing the tokenizer, retriever, and model, and defining a function to generate responses.

```

What should I implement next?
Untitled-1.ipynb • Cell 1:1-33
GitHub Copilot
Next, you should set up the tokenizer, retriever, and model for the RAG system. Here is a step-by-step plan:
1. Initialize the Tokenizer: Load the tokenizer for the RAG model.
2. Initialize the Retriever: Set up the retriever with the necessary configurations.
3. Initialize the RAG Model: Load the RAG model for sequence generation.

Here is the code to implement these steps:

# Initialize the tokenizer
tokenizer = RagTokenizer.from_pretrained("facebook/rag-sequence-nq")
# Initialize the retriever
retriever = RagRetriever.from_pretrained("facebook/rag-sequence-nq")
# Initialize the RAG model
model = RagSequenceForGeneration.from_pretrained("facebook/rag-sequence-nq")

def generate_response(question):
    # Example usage
    # question = "What is the capital of France?"
    # response = generate_response(question)
    # print(response)

    # Expand on this by integrating the retriever with your own dataset and fine-tuning the model if needed.

```

Figure 8: Code generation from prompt using Copilot Chat

B. Code Debugging and explanation GitHub Copilot streamlines debugging by analyzing code, interpreting errors, and offering solutions. It understands exceptions, call stacks, frames, and variable values, acting as a debugger-aware AI. This means it can provide context-specific insights into error messages, variables, and call stack details, helping developers identify and resolve issues efficiently. Using the "Ask Copilot" feature, developers can inquire about code specifics, including call stacks, exceptions, and breakpoints, without manually sharing context. Copilot also suggests

using conditional breakpoints and tracepoints for effective error detection, integrating seamlessly within the IDE to enhance debugging workflows.

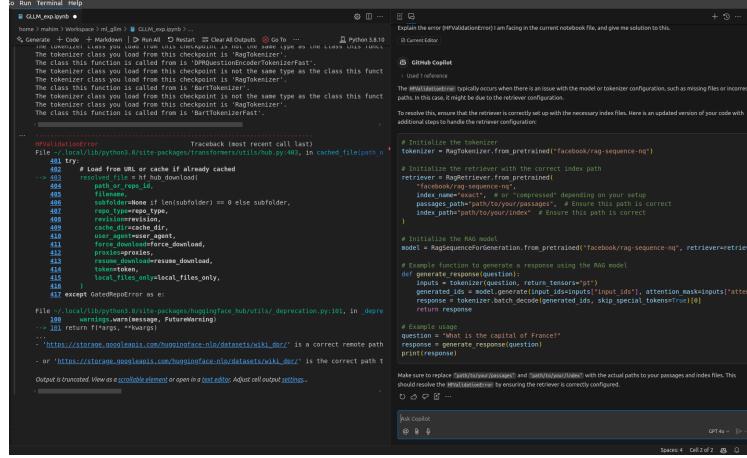


Figure 9: Error solution suggested by Github Copilot

The user must highlight the appropriate code snippet in the integrated development environment (IDE) in order to use GitHub Copilot for code explanation. Once highlighted, the user can choose the "Explain Code" option from the Copilot context menu or sidebar. The functionality of the code and its place in the program's context will subsequently be explained in a natural language description produced by Copilot.

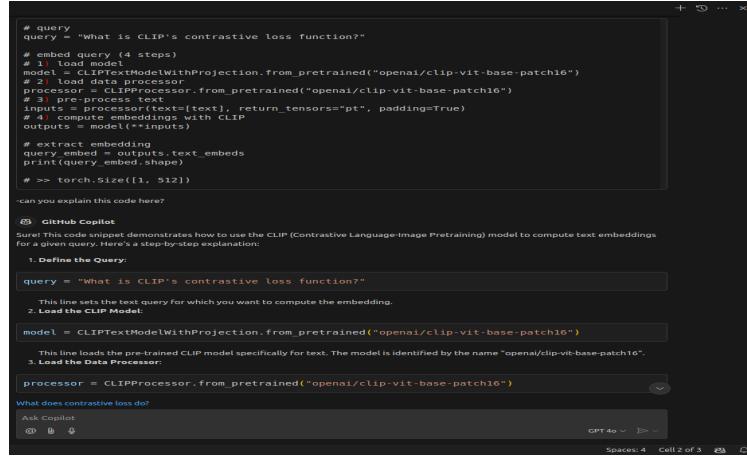


Figure 10: Code explanation given by GitHub Copilot

3.2 ChatGpt

ChatGPT is a conversational platform that uses OpenAI's cutting-edge language models to generate text, Question Answering, and assist with tasks. It is intended to provide natural, human-like interactions, making it useful for a wide range of tasks, from writing assistance to problem-solving. It is accessible directly through a web browser.

3.2.1 Evaluation

ChatGPT faces various security threats due to its wide use in different applications[43]. A key risk is prompt injection, where attackers manipulate inputs to access sensitive information or bypass content filters [53][21] [38] [49][35]. Another danger is data poisoning, which is one of the threats; it involves adding malicious data during training, resulting in biased or harmful results[48][46][42]. Another critical issue is model inversion, which allows attackers to retrieve sensitive information from training data and raises privacy concerns [3]. Additionally, adversarial attacks trick the model into generating incorrect or harmful responses, while privacy breaches can leak personal or confidential information,

especially in sensitive environments [29]. An essential risk is unauthorized access, which helps attackers control the system, change responses, or steal data, posing serious security risks. Another threat is output manipulation, which involves changing ChatGPT's responses to spread false information or achieve harmful objectives [6]. A primary drawback is that bias amplification reinforces social biases present in the training data, leading to unfair or discriminatory responses. Malicious fine-tuning involves retraining ChatGPT on harmful data, inserting hidden vulnerabilities, and compromising its security. ChatGPT often encounters hallucination issues in code generation, where it confidently produces incorrect or non-existent information [43]. Common problems include dead or unreachable code, which leads to inefficiencies and unused code paths. Syntactic errors, where the code is grammatically incorrect, cause compilation or execution failures [3]. Logical errors result in incorrect functionality, making the output behave unexpectedly. Robustness issues arise when the code fails to handle edge cases or unexpected inputs, leading to crashes[28]. Moreover, hallucinations can introduce security vulnerabilities, creating exploitable weaknesses that may compromise data or system integrity [13].

3.2.2 User Feedback Overview

It was observed in our survey that ChatGPT saves 15 to 30 minutes on code review and refactoring tasks and up to 35 minutes on research and documentation efforts. Users also rated its generated code quality, with 15.9% rating it at 80% and 13.6% rating it at 85%. Despite these strengths, challenges persist, including susceptibility to adversarial attacks, data leaks, and hallucinations. Issues such as syntactic and logical errors in generated code emphasize the need for critical evaluation by developers. ChatGPT's error identification capabilities received an average rating of 3.85 out of 5, with 76.7% of users reporting enhanced collaboration and code-sharing capabilities.

3.2.3 Security Analysis

ChatGPT uses an open-source library called Redis to store user data. Hackers exploited this weakness and were able to access chat histories. If a user's request is canceled after reaching the first queue but before the response is sent to the second queue, it will be directed to the next person with a similar question. ChatGPT Plus users who were active during the breach were the main victims, and OpenAI informed those believed to be affected [21]. ChatGPT has experienced data leaks, including a 2023 breach by OpenAI that exposed 1.2% of ChatGPT Plus users' data for nine hours [48]. Concerns have been raised about leaks involving conversations, personal data, and login credentials, potentially from hacker attacks or privacy policy violations[46]. ChatGPT data leaks happen when sensitive information is accidentally or intentionally shared with unauthorized people. Users may accidentally type confidential details into the chat, or there may be weaknesses in the handling and security of the data [42]. Sensitive information can include personal details such as names, addresses, phone numbers, Social Security numbers, financial data such as bank account or credit card numbers, login credentials, and even health-related information. The main risk comes from human error, where users may unknowingly share such personal or financial information, thinking the chat is safe. This can lead to identity theft, financial loss, legal trouble, and damage to personal or company reputation. To reduce the risk of data leaks, users should avoid sharing sensitive information, use strong passwords to protect their accounts, and consider using tools like the SURF Security Enterprise Browser. This browser helps keep data safe by controlling how it moves, blocking exposure, keeping detailed activity logs, and securing login information.

3.2.4 Case study

A. Basic Code Generation

Prompt: "Write JavaScript to sort an array of numbers." Chat Link

```
// Javascript code for Array sort
let numbers = [5, 3, 8, 1, 2];
numbers.sort((a, b) => a - b);
console.log(numbers);
```

Figure 11: Basic Code Generation ChatGpt Reply

B. Adaptive Code Generation

ChatGPT not only generates code based on specific instructions but also understands the broader context of the project. *Prompt:* "Write a function that calculates the factorial of a number." Chat Link

```

python
def factorial(n):
    """Calculate the factorial of a non-negative integer n."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Example usage
number = 5
print(f"The factorial of {number} is {factorial(number)}")

```

Figure 12: Adaptive Code Generation ChatGpt Reply

C. Contextual Understanding of Edits

ChatGPT can comprehend the entire flow of code and suggest relevant changes based on past interactions.
Prompt: "Refactor the functions for efficiency." Chat Link

```

# Simple Item Management App
# List to hold items
items = []

# Function to add an item
def add_item(item):
    items.append(item)
    print(f"Added item: {item}")

# Function to remove an item
def remove_item(item):
    if item in items:
        items.remove(item)
        print(f"Removed item: {item}")
    else:
        print(f"Item '{item}' not found.")

# Function to display all items
def display_items():
    print("Current items:")
    for i in range(len(items)):
        print(f"{i + 1}: {items[i]}")

# Function to clear all items
def clear_items():
    global items
    items = []
    print("All items cleared.")

# Using global to modify the items list
items = []
print("All items cleared.")

# Function to search for an item
def search_item(item):
    if item in items:
        print(f"Item '{item}' found.")
    else:
        print(f"Item '{item}' not found.")

# Example usage
add_item("Apple")
add_item("Banana")

display_items()
search_item("Apple")
remove_item("Banana")
display_items()
clear_items()
display_items()

```

Figure 13: Contextual Understanding Input Example

```

# Simple Item Management App
# List to hold items
items = []

# Function to add an item
def add_item(item):
    items.append(item)
    return f"Added item: {item}"

# Function to remove an item
def remove_item(item):
    try:
        items.remove(item)
        return f"Removed item: {item}"
    except ValueError:
        return f"Item '{item}' not found."

# Function to display all items
def display_items():
    return "\n".join(f"{i + 1}: {item}" for i, item in enumerate(items))

# Function to clear all items
def clear_items():
    global items # Using global to modify the items list
    items.clear()
    return "All items cleared."

# Function to search for an item
def search_item(item):
    if item in items:
        return f"Item '{item}' found."
    return f"Item '{item}' not found."

# Example usage
if __name__ == "__main__":
    print(add_item("Apple"))
    print(add_item("Banana"))
    print(display_items())
    print(search_item("Apple"))
    print(remove_item("Banana"))
    print(display_items())
    print(clear_items())
    print(display_items())

```

Figure 14: ChatGpt Reply Contextual Understanding

D. Efficient Multi-Line Edits

ChatGPT can handle formatting, code restructuring, and inserting relevant comments, managing multiple adjustments simultaneously.

Prompt: "Update all my print statements to use Python's logging module instead." Chat Link

E. Next-Step Anticipation

ChatGPT often anticipates what you might need next based on the previous commands or coding patterns, such as by suggesting the next logical function, structure, or implementation.

Prompt: " If write a class for handling database connections, ChatGPT might automatically suggest methods for closing connections, handling exceptions, or creating queries." Chat Link

F. Iterative Debugging and Optimization

When debugging code, ChatGPT is not merely reactive; it offers proactive suggestions based on detected errors or inefficiencies in code.

```
-- Next-Step Anticipation Input
class DatabaseConnection:
    def __init__(self, db_name):
        self.db_name = db_name
        self.connection = None
    def connect(self):
        """Code to connect to database
        pass
-- Next-Step Anticipation ChatGPT Reply
import sqlite3

class DatabaseConnection:
    def __init__(self, db_name):
        self.db_name = db_name
        self.connection = None
    def connect(self):
        """Establish a connection to the database."""
        try:
            self.connection = sqlite3.connect(self.db_name)
            print("Connected to database: " + self.db_name)
        except sqlite3.Error as e:
            print(f"An error occurred while connecting to the database: {e}")
    def close(self):
        """Close the database connection."""
        if self.connection:
            self.connection.close()
            print("Database connection closed.")
# Example usage:
db = DatabaseConnection("my_database.db")
db.connect()
# Perform database operations here...
db.close()
```

Figure 15: Next-Step Anticipation Input and ChatGPT Output

```
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Example of previous print statements
# print("Starting the process")
logging.info("Starting the process")

# print("An important value:", important_value)
logging.info("An important value: %s", important_value)

# print("An error occurred")
logging.error("An error occurred")

# print("Process completed")
logging.info("Process completed")
```

Figure 16: Adaptive Code Generation Example

Prompt: "Why is my code running slowly?".

ChatGPT Reply: "After analyzing your code, ChatGPT might suggest optimizations such as reducing the time complexity of a nested loop or replacing a recursive function with an iterative approach to improve performance."

G. Enhancing Code Quality and Best Practices

ChatGPT helps to enforce best practices for writing clean, readable, and efficient code. It emphasizes the importance of well-structured documentation, meaningful variable names, and modular code design to improve overall code quality.

Prompt: "How can I improve this function?".

ChatGPT Reply: "In addition to refactoring the code for better efficiency, ChatGPT may suggest adding docstrings, meaningful comments, and adopting consistent naming conventions to enhance the function's readability and maintainability.

H. Third-Party API Integration ChatGPT aids in seamlessly integrating third-party APIs into projects, providing examples of how to authenticate, send requests, and handle responses.[Chat Link](#)

Example: "Integrating a weather API?".

Prompt: "How can I fetch weather data from an external API?".

I. Error Correction and Smart Rewrites ChatGPT automatically detects and suggests corrections for minor syntax mistakes and typing errors, refining code effectively. It excels at pinpointing errors within code by analyzing the syntax and logic. It can recognize issues that might be overlooked during manual reviews, helping developers maintain code quality [Chat Link](#).

Example: "If you provide a function that has a logical error, such as not handling edge cases, ChatGPT can help identify that?".

Prompt: "What's wrong with this code?".

Once errors are identified, ChatGPT provides specific corrections tailored to the context of your code. This includes syntax corrections, logical fixes, and optimizations.

```

import requests

def get_weather(city):
    # Replace 'your_api_key' with your actual OpenWeatherMap API key
    api_key = "your_api_key"
    base_url = "http://api.openweathermap.org/data/2.5/weather"

    # Construct the complete API URI
    complete_url = f"{base_url}?q={city}&appid={api_key}&units=metric"

    # Send a GET request
    response = requests.get(complete_url)

    # Check if the request was successful
    if response.status_code == 200:
        data = response.json() # Parse the JSON response
        main = data['main'] # Get main data
        weather = data['weather'][0] # Get weather description

        # Extract desired information
        temperature = main['temp']
        humidity = main['humidity']
        description = weather['description']

        print(f"City: {city}")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Weather: {description.capitalize()}")
    else:
        print("City not found!")

# Example usage
get_weather("London")

```

Figure 17: Third-Party API Integration Example

```

-- Correcting Errors Input
def divide(a, b):
    return a / b

-- Suggestions Correcting Errors by ChatGPT Reply
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."

```

Figure 18: Correcting Errors Input and Chatgpt Reply

Example:" If your code snippet is meant to calculate the sum of a list but fails, ChatGPT can provide corrections." Chat Link

J. Suggest alternative methods for Code Beyond fixing specific errors, ChatGPT suggests alternative methods and best practices for error handling and code optimization.Chat Link.

Example: "Suggesting a try-except block for error handling?".

K. Generate Test Cases for Unit Testing

ChatGPT not only highlights errors but also recommends structured approaches to resolving them. This could involve: Chat Link

- Breaking down complex functions into simpler ones.
- Suggesting unit tests to ensure correctness.

Example: "Suggesting a try-except block for error handling?"

L. Code Refactoring and Optimization

ChatGPT can Refactoring the code as well as can Optimize the code. Chat Link

Example: "If you provide a nested loop for searching through a list, ChatGPT might suggest using a more efficient algorithm, like binary search or a hash table for faster lookups.?"

M. Suggest New Frameworks or Libraries

ChatGPT is a helpful tool for providing code examples in different frameworks and libraries, making coding easier for developers. It can generate code, help debug issues, and explain complex programming concepts. This is useful for both beginners and experienced programmers, as it simplifies coding and helps them learn new technologies faster. Chat Link

Example: "If you want to switch a web project from Flask to FastAPI, ChatGPT can translate your code and explain how to use the newer framework?"

In many organizations, legacy codebases require updates to align with modern frameworks and practices. ChatGPT helps migrate legacy code by suggesting best practices, refactoring strategies, and identifying deprecated functions.

N. Converting a legacy Flask app to a newer version

ChatGPT is Updating an old Flask application to a newer version requires careful planning to maintain its functionality and quality. The process can follow best practices from software modernization frameworks, which focus on preserving the application's core value while upgrading the technology stack. Chat Link

Example: "How can I update this endpoint for Flask 2.0?"

```
-- Refactoring Monolithic Code to Microservices
# Monolithic User Authentication
class UserAuth:
    def login(self, username, password):
        if username == 'admin' and password == 'password':
            return True
        return False

-- Refactoring Monolithic Code to Microservices Input
# Microservice for User Authentication
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/auth', methods=['POST'])
def authenticate():
    data = request.json
    username = data.get('username')
    password = data.get('password')
    if username == 'admin' and password == 'password':
        return jsonify({"status": "success"}), 200
    else:
        return jsonify({"status": "failure"}), 401

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Figure 19: Converting a legacy Flask app to a newer version Input and Chatgpt Reply

O. Converting an Android RecyclerView to an iOS UITableView

ChatGPT Converting an Android RecyclerView to an iOS UITableView requires understanding the differences between the two UI frameworks. Using automated tools can make the migration process easier. By applying cross-platform migration technique. Chat Link

Example: How do I convert this RecyclerView code into its iOS equivalent?"

P. Refactoring Monolithic Code to Microservices

ChatGPT can guide you through splitting services, managing API communication, and ensuring scalability.

Example: "Extracting a user authentication service from a monolithic system?" *Prompt:* "How can I refactor user authentication into a microservice?"

Q. Database Query Optimization

```
-- Converting an Android RecyclerView to an iOS UITableView Input
RecyclerView recyclerView = findViewById(R.id.recyclerview);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
recyclerView.setAdapter(new CustomAdapter(dataList));

-- Converting an Android RecyclerView to an iOS UITableView ChatGPT Reply

import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {

    let tableView = UITableView()
    let dataList = ["Item 1", "Item 2", "Item 3"] // Replace with your data

    override func viewDidLoad() {
        super.viewDidLoad()

        // Set up the table view
        tableView.frame = self.view.bounds
        tableView.delegate = self
        tableView.dataSource = self

        // Register a custom cell if needed
        tableView.register(UITableViewCell.self, forCellReuseIdentifier: "cell")

        // Add the table view to the view hierarchy
        self.view.addSubview(tableView)
    }

    // MARK: - UITableViewDataSource methods

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return dataList.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell", for: indexPath)
        cell.textLabel?.text = dataList[indexPath.row] // Replace with your custom data
        return cell
    }
}
```

Figure 20: Converting an Android RecyclerView to an iOS UITableView Input and Chatgpt Reply

```
-- Converting a legacy Flask app to a newer version Input
from flask import Flask, request

app = Flask(__name__)
@app.route('/old-endpoint', methods=['GET'])

def old_endpoint():
    return "This is the old endpoint"

-- Converting a legacy Flask app to a newer version
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/new-endpoint', methods=['GET'])
def new_endpoint():
    # Optionally, you can get query parameters
    param = request.args.get('param', default=None)

    # Return a JSON response
    response = {
        "message": "This is the new endpoint",
        "param": param
    }
    return jsonify(response)

if __name__ == '__main__':
    app.run(debug=True)
```

Figure 21: Refactoring Monolithic Code to Microservices Input and Chatgpt Reply

Optimizing database queries for performance can be challenging. ChatGPT can assist developers in writing optimized SQL queries or suggest indexing strategies.

Example: "Optimizing a SQL query for a large dataset?" *Prompt:* "How can I optimize this query for faster performance?"

```
-- Database Query Optimization Input
SELECT * FROM orders WHERE customer_id = 1234 AND order_date > '2023-01-01';

-- Suggested Optimized Query
CREATE INDEX idx_customer_order ON orders(customer_id, order_date);

SELECT order_id, order_total, order_date
FROM orders
WHERE customer_id = 1234 AND order_date > '2023-01-01';
```

Figure 22: Database Query Optimization Input and Chatgpt Reply

3.3 Cursor AI

Cursor is an AI code editor that enhances productivity by anticipating edits and providing intelligent coding suggestions. It is a fork of VS Code. This allows us to focus on making the best way to code with AI while offering a familiar code editing experience. It prioritizes privacy with a local storage mode for code and integrates effortlessly with existing tools and workflows.

3.3.1 Evaluation

AI-assisted coding tools like **Cursor AI** have revolutionized programming by enabling natural language-driven code generation, debugging, and optimization. However, they pose risks, particularly in terms of security vulnerabilities. Since they are trained on vast datasets, including potentially insecure or outdated code, they can generate vulnerabilities like SQL injection, XSS, weak authentication methods, and poor error handling [1]. Developers must follow security best practices to avoid deploying unsafe code. Users have reported challenges with Cursor AI when using multiple files [27]:

- **File Version Hallucinations:** AI may mistake current file versions for outdated ones, leading to redundant or incorrect suggestions [37].
- **Contextual Gaps:** Without full folder analysis, AI struggles to understand multi-file contexts, leading to irrelevant or repetitive suggestions [37].

These issues highlight inefficiencies in the tool's management of file context, affecting usability and performance. Additionally, misuse of Cursor AI can degrade code quality. While the generated code may appear correct initially, small errors and inefficiencies can accumulate, undermining project quality. Cursor AI also has limitations, particularly with complex projects [20]. While it handles simple tasks well, it struggles with larger project structures, leading to incorrect suggestions and subtle bugs. Although effective for basic code, it delays with advanced needs, such as specific business logic or custom frameworks [20], creating a false sense of security for new developers who may rely too heavily on AI-generated code. Another concern is data privacy, especially for projects involving sensitive information. Despite Privacy Mode, users have raised questions [27] about whether the tool still stores data, potentially violating NDAs. Enabling "Privacy Mode" ensures that no code is stored, except for temporary prompt data retained by OpenAI and Anthropic for 30 days [4].

3.3.2 User Feedback Overview

A user survey indicated that 15.2% of users rated its generated code quality at 50%, with 9.1% rating it between 60% and 80%. Cursor AI's error identification capabilities were found to effectively reduce debugging time. Additionally, 69.7% of users reported enhanced collaboration, though 18.2% found it ineffective for collaborative tasks. Reliability and performance ratings showed that 48.5% of users gave it a 4, while 33.3% rated it a 3.

3.3.3 Security Analysis

Cursor AI uses subprocessors (e.g., AWS, Fireworks, OpenAI, Anthropic, Google Cloud Vertex API) and cloud services to deliver its AI features[5]. When we disable privacy mode, Cursor AI gathers telemetry and usage data, such as code snippets and editor actions, to enhance its AI capabilities[4]. Cursor AI temporarily caches and encrypts this data on servers, but neither permanently stores nor uses it for training purposes. However, if privacy mode is enabled, no code data is stored or retained by Cursor or any third party[4]. When using an API key, requests pass through Cursor's backend for the final prompt construction. When enabling code indexing, Cursor uploads small portions of code for embedding calculations and deletes the raw code after completing the process. Cursor stores only the embeddings and associated metadata, including file names and hashes [37]. Furthermore, file contents are temporarily cached on servers, encrypted with unique client keys, and are not utilized for training when Privacy Mode is enabled[20].

3.3.4 Case study

A. Code Generation

Cursor predicts your next steps based on recent changes, tracks the codebase, and suggests relevant code, enhancing development efficiency. It adapts to past interactions, improving the accuracy of its suggestions and ensuring context-aware edits.

B. Multi-Line Edits Cursor suggests multiple edits at once, saving time and reducing errors. It intelligently rewrites code blocks for better readability and performance, streamlining the development process.

```
You, 9 seconds ago | 1 author (You) | 0 implementations
7 pub struct EncodedMessageQueue {
6     pub queue: Vec<(>EncodedMessage, ClientFilter)>,
5
4     sender: Arc<Sender<Vec<(>EncodedMessage, ClientFilter)>>>,
3     receiver: Arc<Receiver<Vec<(>EncodedMessage, ClientFilter)>>>,
2 }
1
6 impl |EncodedMessageQueue { You, 8 seconds ago + Uncommitted changes
5     pub fn new() -> Self {
4         let (sender, receiver) = crossbeam_channel::unbounded();
3         Self {
2             queue: vec![],
1             sender: Arc::new(sender),
             receiver: Arc::new(receiver),
         }
        }
    }
}
```

Figure 23: Code Generation by cursor

```
.is_see_through(true) BlockBuilder
.light_reduce(true) BlockBuilder
.transparent_standalone(true) BlockBuilder
.build(),
Block::new(name: "Orange Concrete Block").id(5006).build(),
Block::new(name: "Blue Concrete Block").id(5007).build(),
Block::new(name: "Red Concrete Block").id(5008).build(),
Block::new(name: "White Concrete Block").id(5009).build(),
Block::new(name: "Ivory").id(5012).build(),
make_stairs(name: "Oak Stairs", id: 5013),
make_stairs(name: "Ivory Stairs", id: 5014),
make_stairs(name: "Stone Stairs", id: 5015),
]);
});
```

Figure 24: Multi-Line Edits by cursor

C. Smart Rewrites Cursor fixes errors in your code, improving readability and ensuring consistency in coding style. It refactors your code for better performance, helping to prevent potential bugs and save time on corrections.

```
entitiesRef.current = entities;

const debug = new Debug(document.body, {
  dataStyles: [
    top: '10px' left: '10px' fixed,
    zIndex: '1000',
    color: '#fff',
    backgroundColor: 'var(--color-overlay)',
    padding: '8px',
    fontSize: '12px',
    border: '2px solid var(--color-text-primary)',
  ],
});

const gui = new GUI();
```

Figure 25: Smart Rewrites by cursor

D. Cursor Prediction By predicting your next cursor position, Cursor enhances navigation and streamlines coding. It anticipates movements, making it easier to navigate a large codebase and improving workflow efficiency.

E. Identifying Errors and Fix Suggestions

Cursor AI can identify errors effectively. In the compile time, it gives the suggestion by AI which is an important feature in cursor AI.

Cursor Composer is an advanced AI feature in the Cursor editor that simplifies multi-file editing and full application development. It allows developers to provide high-level instructions for building or modifying entire applications while accounting for the project structure. Its key features include multi-file editing, app generation, contextual understanding,

```

8   const unbindKeyC = inputs.bind("KeyC", () => this.toggle(), namespace, {
7     identifier: Perspective.INPUT_IDENTIFIER,
6     checkType: "code",
5   });
4
3   const unbindKeyF = inputs.bind("KeyF", () => this.toggle(true), namespace, {
2     identifier: Perspective.INPUT_IDENTIFIER,
1     checkType: "code",
160   });
1
2   // no clue why but this seems to work, f5 seems to be reversed
3   const unbindF5 = inputs.bind("F5", () => this.toggle(true), namespace, {
4     identifier: Perspective.INPUT_IDENTIFIER,
5     checkType: "code",
6   });
7
8   this.inputs = inputs;
9
10  return () => {
11    try {
12      unbindKeyC();
13      unbindF5(); [← tab]
14    } catch (e) {
15      // Ignore.
16    }
17  };
18

```

Figure 26: Cursor Prediction

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ahmed@ahmed-HP-ZBook-Studio-G5:~/Projects/XELS/XELS$ source xelsen/bin/activate
(xelsenv) ahmed@ahmed-HP-ZBook-Studio-G5:~/Projects/XELS/XELS$ python src/testAI.py
Traceback (most recent call last):
  File "/home/ahmed/Projects/XELS/XELS/src/testAI.py", line 21, in <module>
    KeyboardInterrupt

(xelsenv) ahmed@ahmed-HP-ZBook-Studio-G5:~/Projects/XELS/XELS$ python src/testAI.py
Traceback (most recent call last):
  File "/home/ahmed/Projects/XELS/XELS/src/testAI.py", line 2, in <module>
    print
NameError: name 'printf' is not defined. Did you mean: 'print'?
(xelsenv) ahmed@ahmed-HP-ZBook-Studio-G5:~/Projects/XELS/XELS$ 0

```

Figure 27: Identifying Errors and Fix Suggestions

and interactive refinement. Cursor integrates custom API keys for enhanced flexibility and provides effective error resolution with interactive fixes, ensuring maintainable and functional code.

3.4 Codeium AI

Codeium AI is a cutting-edge AI-driven tool that revolutionizes software development. It improves code quality, automates testing processes, and integrates effortlessly with popular coding platforms such as VSCode and JetBrains IDEs.

3.4.1 Evaluation

Some companies avoid AI tools because of security concerns, and developers with unique workflows may have trouble using these assistants. Codeium solves these problems by allowing on-premise deployment, making it secure and customizable for specific projects. This ensures that developers can maintain control over their code while using the tool. Codeium is also fully integrated and works with other development tools, so it can be used alongside existing workflows without risk. By offering local deployment, Codeium ensures that sensitive data stays within the company, reducing the chances of outside security breaches [53][36]. This feature also helps companies meet legal requirements for data privacy and security [14]. Additionally, the tool can be customized to fit specific coding practices, which ensures the AI works according to company standards. With over 300,000 free users and 100 enterprise clients, Codeium is widely trusted for its combination of productivity and security [3][6][40]. It enables developers to work faster while ensuring their data is safe.

3.4.2 User Feedback Overview

Surveys revealed that Codeium significantly streamlined workflows, with time savings highlighted as a major benefit. Its generated code quality was rated with peaks at 40% and 60%, each at 12.5%, while higher ratings above 75% were relatively scarce. Debugging time reduction was another strength, as users found it effective for resolving complex issues. Collaboration and support metrics indicated that 57.1% of users experienced improved collaboration, though

32.1% reported it as ineffective for teamwork. Reliability and performance ratings showed that 58.1% of users rated it a 3, with 16.1% and 6.5% giving it ratings of 4 and 5, respectively.

3.4.3 Security Analysis

Codeium AI [40] processes various types of data, including code snippets, metadata, user authentication details, and model configurations, which are transmitted through its system using cloud infrastructure. This raises concerns about data security and potential leaks, such as the risk of personal data embedded in code, cross-border data transfers, unauthorized access to source code, AI-generated vulnerabilities, and intellectual property violations. Codeium mitigates these risks by implementing strict code review guidelines, employing contractual safeguards for cross-border transfers, using advanced user authentication and access control, conducting vulnerability scanning, and ensuring proper license verification. To prevent data leaks, Codeium [14] emphasizes encryption, offers self-hosted deployment options for enterprise users, and restricts data access to authorized personnel. Through these measures, Codeium aims to ensure responsible data handling and privacy protection, urging organizations to implement continuous monitoring and internal controls.

3.4.4 Case study

A. Code Generation Codeium's code generation feature enables code generation simply by describing tasks in natural language. Using natural language processing, it creates high-quality code that matches with needs. It works for various programming languages, such as Go, HTML, or Unity. This feature makes coding faster and more accessible, even for complex tasks. A best-in-class proprietary model, trained from scratch to optimize for speed and accuracy, powers this feature.

Figure 28: CodeiumAI Code Generation

B. Code Refactoring It has strong code refactoring options. There are multiple options and features for code refactoring, which are shown below.

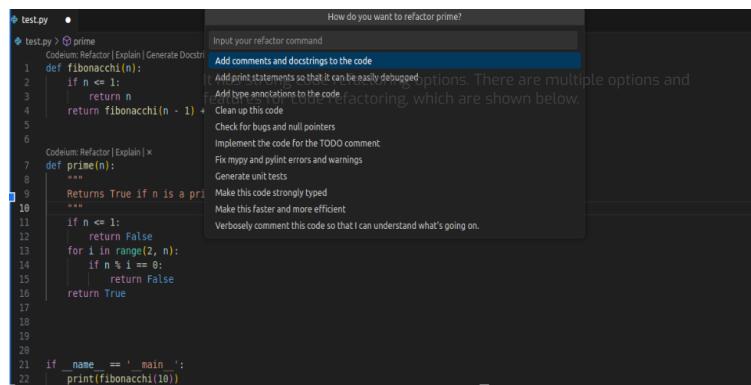


Figure 29: CodeiumAI Code Refactoring

C. Code Debugging Codeium offers real-time debugging through its integrated chat support. It leverages advanced AI models for precise code analysis and bug detection. Supports debugging across 70+ programming languages, ensuring compatibility with diverse projects. Enhances debugging with smart autocomplete suggestions tailored to resolving issues more efficiently.

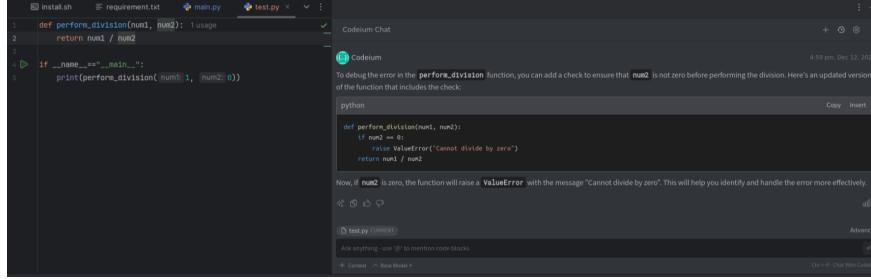


Figure 30: CodeiumAI Code Debugging

D. Code explanation Codeium’s code explanation feature helps users understand code easily by providing clear, concise explanations. It uses natural language generation and code analysis to explain the purpose, logic, and meaning of code snippets or expressions. Users can input code in various languages like Java, R, Python, or C# to get instant explanations. Codeium supports multiple programming languages and editors. To use the feature, press **Ctrl+Shift+Space** or click the Codeium icon in your editor. A window will appear where you can paste or select code, and Codeium will generate an explanation. This feature makes coding easier.

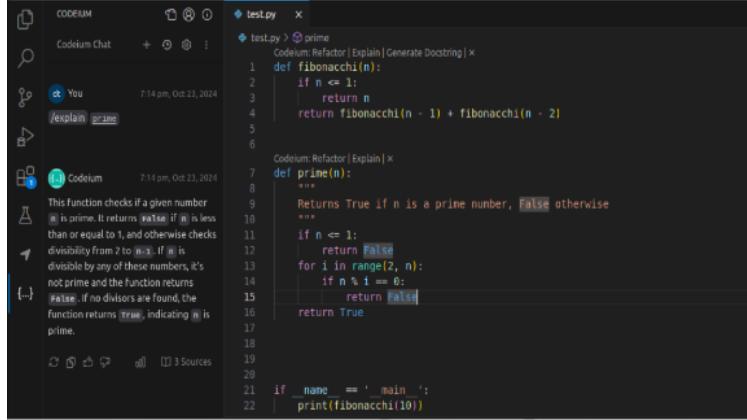


Figure 31: CodeiumAI Code explanation

E. Code Completion

Codeium’s code completion feature helps users write code faster by suggesting relevant keywords, functions, and parameters. It uses deep learning to provide context-aware suggestions based on code. For example, in Python, it suggests correct syntax and arguments; in SQL, it recommends tables and columns; in Excel, it offers suitable functions and formats. This feature supports multiple programming languages and editors.

F. Supercomplete Features In October 2024, Codeium introduced a new feature called Supercomplete. Supercomplete is a passive AI that shows the changes insertions, deletions, and edits that match your next action in a pop-up next to the text in your editor. It works independently of where your cursor is positioned.

4 LLM Security Issue

LLM security is a comprehensive set of practices that are designed to protect large language models from potential threats and vulnerabilities [26]. To protect this security, regular code reviews and concurrent programming play a crucial role. Additionally, we can reduce security risks by using OWASP guidelines and using trusted libraries. Furthermore, to block malicious data and enhance security, implementing user input validation and sanitization processes is very essential. Data encryption, employing methods such as AES and TLS, ensures data security and safeguards any

```

def main(input_directory, output_directory): # rafi-sol *
    os.makedirs(output_directory, exist_ok=True)

    pdf2text = Pdf2Text()

    # Iterate through all files in the input directory
    for filename in os.listdir(input_directory):
        if filename.endswith('.pdf'):
            # Construct full paths for input and output files
            input_file_path = os.path.join(input_directory, filename)
            output_file_name = os.path.splitext(filename)[0] + '.txt'
            output_file_path = os.path.join(output_directory, output_file_name)

            # Parse the PDF and save the text output
            pdf2text.extract_and_combine_text_langchain_PypdfLoader(input_file_path, output_file_path)
            print(f'Processed: {filename} > {output_file_name}')

    # Iterate through all files in the output directory
    for filename in os.listdir(output_directory):
        if filename.endswith('.txt'):
            # Construct full paths for input and output files
            input_file_path = os.path.join(output_directory, filename)
            output_file_name = os.path.splitext(filename)[0] + '.json'
            output_file_path = os.path.join(output_directory, output_file_name)

```

Figure 32: CodeiumAI Code explanation

systems from unauthorized access. Furthermore, we need to maintain security logs and monitor suspicious activity to significantly reduce the attacks. For example,

```

# Insecure Example: Exposing Sensitive Data
from flask import Flask, jsonify, request
app = Flask(__name__)

@app.route('/localhost:5000/user/1', methods=['GET'])
def get_user(user_id):
    user_data = [
        {"id": 1, "name": "Alice", "password": "alice123"},
        {"id": 2, "name": "Bob", "password": "bobpass"}
    ]
    return jsonify(user_data.get(user_id, "User not found"))

#Running the Flask app
if __name__ == "__main__":
    app.run(debug=True)

```

```

# Vulnerability Example: Blind Trust in Input
@app.route('/query', methods=['POST'])
def query():
    data = request.json
    return eval(data['query']) #Dangerous: Directly evaluating input

#Secure Example: Using Safe Parsing Libraries
@app.route('/query' methods=['POST'])
def query():
    data = request.json

```

(a) API Endpoint Vulnerabilities

(b) Transition in Queries from Participant 1031

Figure 33: Examples of LLM Security

This picture 33a conveys the user creation API endpoint has serious vulnerabilities, as it uses proper authentication and MD5 hashing, which is dangerous for security. Additionally, figure 33b shows two prompts from Participant 1031 included for security rather than querying the AI assistant. Deploying firewalls, and intrusion detection systems, and providing ongoing security training for developers are essential for LLM security.

4.1 Key Components of LLM Security Strategy

LLM security focuses on four main areas: data security, model security, infrastructure security, and ethical concerns [16]. Securing these areas involves a combination of standard cybersecurity practices and LLM-specific protections. Data security involves mitigating risks like data leakage, poisoning, and privacy breaches through robust measures, including encryption, access control, and protocols for ensuring data integrity. Model security focuses on challenges such as misinformation, hallucinations, and denial-of-service attacks, advocating for the adoption of authentication protocols, tamper protection, and thorough validation processes. Infrastructure security emphasizes the need for securing hosting environments with firewalls, encryption, and physical safeguards to protect against both digital and physical threats. Ethical considerations address concerns like bias, toxicity, and discrimination, highlighting the importance of ethical guidelines and responsible practices to ensure fairness and accountability. A comprehensive approach to these dimensions is essential for integrating LLMs securely, reliably, and responsibly into various applications. Data security is crucial for protecting sensitive training data, user inputs, and maintaining data integrity [26]. Securing LLMs requires robust access control, encryption, and monitoring to prevent breaches and unauthorized modifications [16]. Infrastructure hosting LLMs must also be safeguarded against cyber threats to mitigate vulnerabilities [29].

20

Compliance with regulations like GDPR and HIPAA is essential to minimize legal risks and uphold organizational reputation in sensitive environments [26].

4.2 LLM code Vulnerabilities

LLM Code Vulnerabilities are security issues or weaknesses that appear in code produced by the Large Language Model [29]. Code problems can occur for a variety of reasons, including technical errors, human error, open-source software (OSS) reuse, and even unexpected zero-day attacks. Some example are discussed below:

```

Prompt
# Vulnerable Example: Unsafe SQL Query
import sqlite3

connection = sqlite3.connect('users.db')
cursor = connection.cursor()

Malicious input
user_input = "1 OR 1=1"
query = f"SELECT * FROM users WHERE id = [{user_input}]"
SQL Injection happens here
cursor.execute(query)
print(cursor.fetchall())

Secure Example: Using Parameterized Queries
#This won't work with parameterized queries
user_input = "1 OR 1=1"
query = "SELECT * FROM users WHERE id = ?;" 
cursor.execute(query, (user_input,))
print(cursor.fetchall())

```

Fixing Vulnerability

WHERE id = ?;

```

// Buffer Overflow Vulnerabilities
#include <stdio.h>
#include <string.h>

void vulnerable_function(char * input) {
    char buffer[10];
    // No bounds checking
    strcpy(buffer, input);
    printf("Buffer contains: %s\n", buffer);
}

int main() {
    char input[100];
    printf("Enter input: ");
    gets(input); // Dangerous function
    vulnerable_function(input);
    return 0;
}

```

Buffer Overflow


```

// Secure Example
#include <stdio.h>
#include <string.h>

void secure_function(char * input) {
    char buffer[10];
    strncpy(buffer, input, sizeof(buffer) - 1); // Bounds checking
    buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate
    printf("Buffer contains: %s\n", buffer);
}

int main() {
    char input[100];
    printf("Enter input: ");
    fgets(input, sizeof(input), stdin);
    secure_function(input);
    return 0;
}

```

Safer Input Function

(a) SQL Injection vulnerabilities

(b) Buffer Overflows vulnerabilities

Figure 34: Examples of LLM Vulnerability

In the first image 34a, this code exposes a SQL injection vulnerability by directly attaching untested data to SQL queries, specifically the username variable. This allows hackers to manipulate the input and execute SQL commands. Additionally, another code 34b snippet demonstrates a buffer overflow vulnerability due to the lack of checks when copying elements from src to dest. LLM sometimes produces code that uses older programming methods and libraries [7], which are not compliant with modern security. This increases the security risk. Although LLMs are designed to generate useful code, sometimes faulty code is regenerated from the training dataset, which weakens the security of the application [29]. On the other hand, malicious users can create malicious code by manipulating input prompts, such as making minor changes to cause errors [16]. Additionally, errors in LLM’s training data also create major vulnerabilities in the code, especially when errors occur systematically [47]. LLM often lacks the deep contextual understanding needed to create secure code, resulting in sometimes irrelevant advice for specific security contexts [7]. Typically, they are not updated with the latest security vulnerabilities and threats, leaving the generated code open to new attacks [16]. Moreover, errors in generated code go undetected due to lack of security checks, which can cause serious security problems [2].

4.3 LLM hallucination

LLM hallucination refers to instances when a language model, such as ChatGPT, generates information that is incorrect, irrelevant, or nonsensical [28]. This phenomenon can lead to misleading or false outputs that may confuse users or propagate misinformation. A comprehensive taxonomy of hallucination types and issues in Generative Large Language Models focusing on errors in code generation [30]. Hallucinations in LLMs occur when models produce incorrect, inconsistent, or nonsensical outputs, undermining functionality and reliability [43] [13]. Key issues include intent conflicts, where generated code misaligns with overall task goals (overall semantic conflicting) or with specific local intent (local semantic conflicting) [52]. Context deviations manifest as logical inconsistencies, repetitive code, dead code, or context mismatches [5] [4]. Knowledge errors arise from misuse of APIs (incorrect API calls, non-existent methods) and undefined or misused identifiers (variable misnaming, undefined references). Further, expression issues involve incorrect constants, faulty logic [29] in loops, conditions, or branches, and redundant copying of input contexts. Additional errors include IO/assert statement errors, such as incorrect function definitions (wrong parameters, return type mismatches) and improper assignments that disrupt logic. Problems with libraries and parameters encompass

missing or unnecessary libraries, incorrect library imports, and mismatched parameters (extra arguments, missing function parameters). Identifier issues include undefined, misused, or duplicated identifiers (name collisions, wrong scope). Other critical types include semantic misalignment, where generated code produces unintended side effects, and efficiency issues, such as generating performance-inefficient code or overuse of computational resources. Additionally, output structuring errors like improper formatting, invalid indentation, or broken comments can lead to reduced code readability. Addressing these hallucination types is essential for improving LLMs by refining training datasets, enhancing contextual comprehension, and integrating robust validation systems to identify and correct hallucinations. Also, LLMs sometimes give nonsensical answers that are completely unrelated to the prompt [28]. For example, if the prompt asks "Thomas Edison was born in 1847" and it answers "Edison was born in 1947" then this creates a conflict. One study found that 14.3% of ChatGPTs had this type of problem. Another problem is that LLMs often generate context-free or random data [12].

4.4 Case study of LLM hallucinations

Figure 35a illustrates performance inconsistencies caused by ill-formed logic in the binary XOR operation, obscuring the intended functionality. On the other hand, Figure 35c highlights contextual inconsistencies, particularly in misunderstanding Python's zero-based indexing when slicing arrays using 'low-1'. Furthermore, Figure 35e presents code repetition, while Figure 35g contains "dead code" that does not contribute to the program's functionality. Similarly, Figure 35i demonstrates cognitive conflict due to incorrect identifier usage, such as 'largest_max_len_len_string', leading to potential misinterpretation of operations. Additionally, Figure 35k, this image reveals two separate hallucinations within the same program.

4.5 LLM Security Attack and Risks

LLM security mainly focuses on protecting the functionality, integrity, and security of data in large language models. It takes various steps to protect the model, the data it uses, and the supporting infrastructure. Model security thereby ensures that the model is protected from malicious attacks and does not provide incorrect or misleading information [34].

Here, 36 provides examples of both a malicious attack and a non-malicious interaction. In the malicious attack example, an attacker injects a prompt designed to elicit a biased response. The potential threats and methods used by attackers to indirectly manipulate large language models integrated into various applications, highlighting the risks posed to multiple stakeholders, including end-users, developers, automated systems, and the integrity of the LLM itself. Injection methods range from passive approaches, such as retrieving sensitive data, to active methods like malicious emails, user-driven injections, and hidden attacks designed to covertly manipulate prompts. Key threats include information gathering, where attackers extract personal data, credentials, or chat content; fraud through phishing, scams, and masquerading; and intrusion, involving persistence, remote control, or malicious API exploitation. Additionally, attackers may spread malware through malicious prompts, prompt-based worms, or traditional software-based methods. Manipulated content, such as incorrect summaries, disinformation, or biased propaganda, and availability attacks, including Denial of Service (DoS) or computational overload, further illustrate the range of risks. This taxonomy underscores the sophisticated techniques used to exploit LLM vulnerabilities, emphasizing the critical need for comprehensive security measures to safeguard against these evolving threats. The tick mark (✓) indicates a feature or issue is supported, while the cross mark (✗) indicates it is not supported.

5 Discussion

This section presents a discussion based on the observations from our comprehensive investigation of the use of LLMs such as GitHub Copilot, ChatGPT, Cursor AI, and Codeium AI into software development.

5.1 Productivity and Efficiency Gains

One of the most notable benefits of LLMs in software development is the substantial increase in productivity. Tools like GitHub Copilot and ChatGPT have been praised for their ability to reduce the time spent on debugging, code generation, and refactoring. According to user feedback, Copilot, for instance, has been instrumental in reducing debugging time by identifying errors more effectively, with users reporting an average rating of 3.85 out of 5 for error identification effectiveness. Similarly, ChatGPT has been lauded for its versatility in generating code, providing quick fixes, and offering detailed explanations, which significantly streamline the development process. However, the efficiency gains are not uniform across all tools. While Copilot and ChatGPT excel in certain areas, Codeium AI and Cursor AI show mixed results. Codeium AI, for example, is effective in automating repetitive tasks and generating standard code

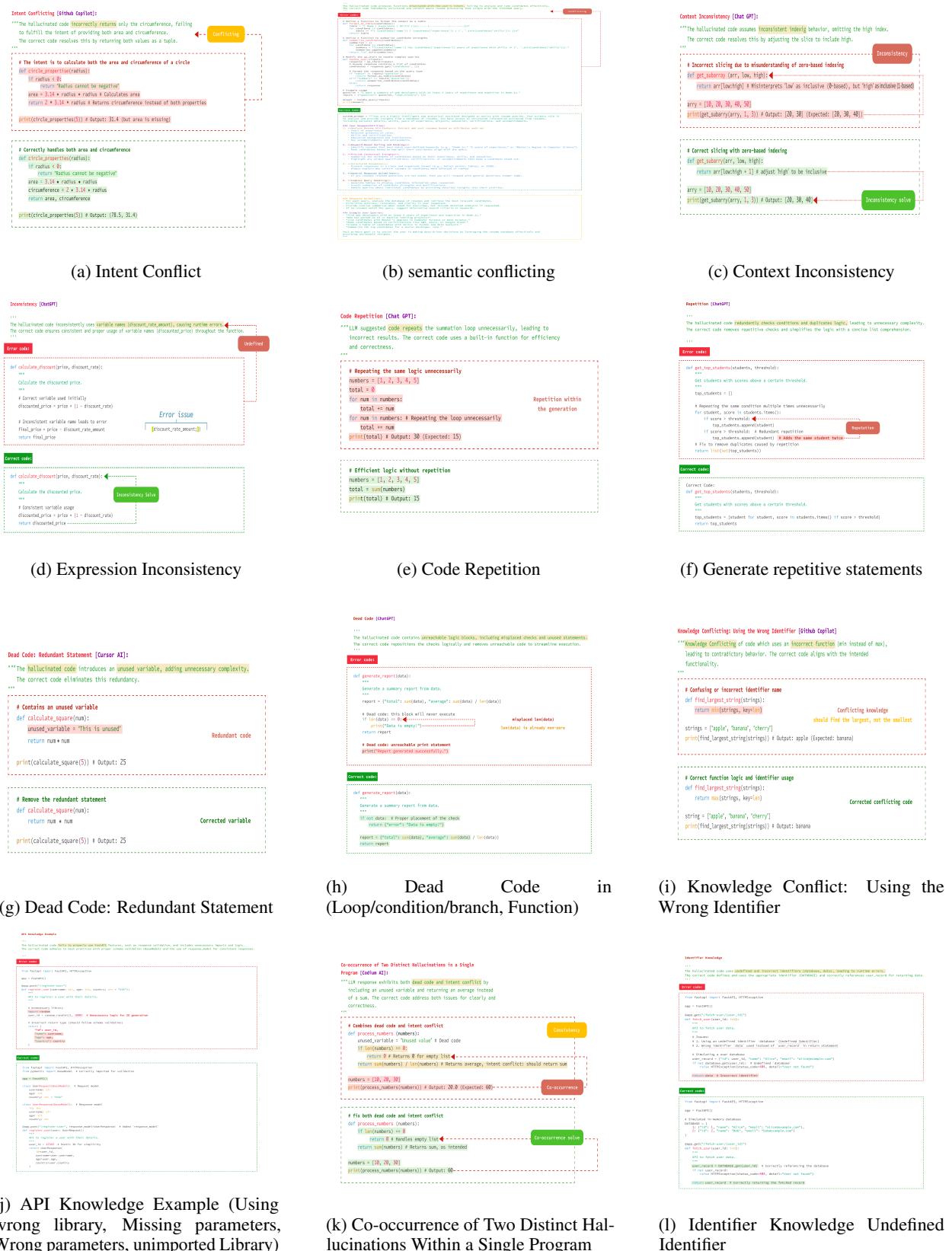


Figure 35: Examples of LLM Hallucination

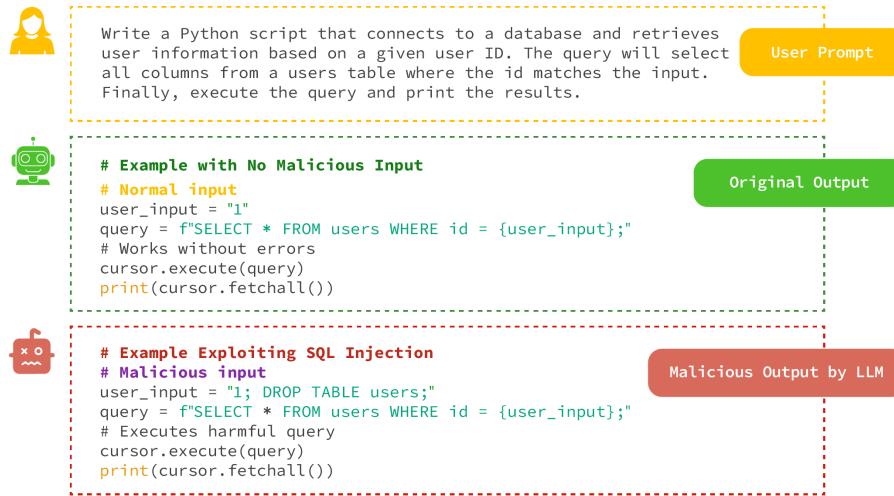


Figure 36: SQL Injection Vulnerabilities in LLM Security

Table 2: Comprehensive Comparison of Features, Issues, and Risks

Feature/Tool	Codeium	ChatGPT	Cursor	Copilot
Replicates Security Vulnerabilities	✓	✓	✓	✓
Insecure Code Suggestions	✓	✓	✓	✓
Hallucinations in Code Generation	✓	✓	✓	✓
Generates Incorrect Syntax/Logic	✓	✓	✓	✓
Mimics SQL Injection Patterns	✓	✓	✓	✓
Context Misinterpretation	✓	✓	✓	✓
Outdated Syntax for Frameworks	✓	✓	✓	✓
Incorrect Library Imports	✓	✓	✓	✓
Bias in Suggestions	✓	✓	✓	✓
Supports Complex Test Generation	✓	✓	✓	✓
Cost	Free	Free/Paid (OpenAI API)	Free/Paid (Pro Plan)	Free/Paid (GitHub Copilot)
Primary Use Case	Code Autocomplete	General Coding Help	Professional Coding	Code Autocomplete
Integration	Lightweight	Standalone	IDE Integration	IDE Integration
Language Support	Multiple	Multiple	Extensive	Multiple
Best For	Free Coding Assistant	General Use	Professional Devs	Professional Devs
Security Risks	✓	✓	✓	✓
Effective for Debugging	✓	✓	✓	✓
Handles Contextual Gaps	Partial Support	Partial Support	Partial Support	Partial Support
Data Privacy Control	✓/✗	✓/✗	✓/✗	✓/✗
Code Snippet Sharing	Partial Support	Partial Support	Partial Support	Partial Support
Automatic Refactoring Support	✓	Partial Support	✓	✓
Community Support & Add-Ons	Growing	Large	Limited	Large

snippets but struggles with more complex, project-specific tasks. This highlights the importance of selecting the right tool for the right task, as the capabilities of these AI tools vary significantly.

5.2 Security Concerns and Vulnerabilities

Despite their productivity benefits, the use of LLMs in code generation introduces significant security risks. One of the most pressing concerns is the replication of existing vulnerabilities. GitHub Copilot, for instance, has been found to propagate insecure coding practices by suggesting code snippets that are vulnerable to SQL injection, cross-site scripting (XSS), and other common security flaws. This is particularly problematic when developers rely heavily on AI-generated code without thorough manual review. Moreover, the risk of data leaks and intellectual property violations cannot be overlooked. Tools like ChatGPT and Codeium AI, which rely on cloud-based infrastructure, pose potential risks of exposing sensitive codebases or user data. For example, ChatGPT's use of Redis for data storage has been exploited in past breaches, leading to unauthorized access to chat histories and user payment information. These incidents underscore the need for robust security measures, including encryption, access controls, and regular security audits, to mitigate the risks associated with AI-generated code.

5.3 Code Quality and Hallucinations

Another critical issue with LLMs is the phenomenon of "hallucination," where the model generates incorrect, irrelevant, or nonsensical code. This is particularly problematic in complex projects where the AI may misinterpret the context or fail to grasp the broader structure of the codebase. For instance, Cursor AI has been reported to struggle with contextual gaps, leading to redundant or incorrect suggestions when analyzing entire folders. Similarly, ChatGPT has been known to suggest non-existent libraries or incorrect function signatures, which can lead to significant errors if not caught during the review process.

The issue of code quality is further compounded by the fact that LLMs often generate syntactically correct but logically flawed code. This creates a false sense of security, as developers may assume that the generated code is functional and secure, only to discover issues later in the development cycle. This highlights the importance of manual code reviews and the use of automated testing tools to ensure that AI-generated code meets the required standards of quality and security.

6 Conclusion

This paper offers a comprehensive exploration of the integration of AI-powered tools in software development, focusing on GitHub Copilot, ChatGPT, Cursor AI, and Codeium AI. Provides an in-depth evaluation of their features, strengths, and weaknesses, incorporating user feedback and security analysis to provide practical insights for developers and organizations. Key findings highlight significant security risks, such as replication of insecure coding practices, data leaks, and vulnerabilities such as SQL injection and cross-site scripting. A major concern is the security risks associated with AI-generated code. LLMs, trained on vast publicly available data, often replicate insecure practices, leading to vulnerabilities. Data leaks and intellectual property violations further complicate their adoption, especially in sensitive environments. Robust measures, including encryption, access controls, and audits, are vital to mitigate these risks. Another critical issue is "hallucination," where LLMs produce incorrect or irrelevant code, particularly in complex projects. Developers must remain vigilant, conducting thorough reviews and testing. By implementing security measures, reviewing code carefully, and adhering to ethical guidelines, the software development community can harness AI effectively while minimizing drawbacks.

References

- [1] Anal A. Hidden dangers of using cursor ai for code generation: What every developer should know, 2024. Accessed: 2024-12-10.
- [2] Sara Abdali, Richard Anarfi, CJ Barberan, and Jia He. Securing large language models: Threats, vulnerabilities and responsible practices. *arXiv preprint arXiv:2403.12503*, 2024.
- [3] Vibhor Agarwal, Yulong Pei, Salwa Alimir, and Xiaomo Liu. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333*, 2024.
- [4] Cursor AI. Cursor ai privacy policy, 2024. Accessed: 2024-12-09.
- [5] Cursor AI. Cursor ai security and vulnerability disclosures, 2024. Accessed: 2024-12-09.
- [6] Moatsum Alawida, Bayan Abu Shawar, Oludare Isaac Abiodun, Abid Mehmood, Abiodun Esther Omolara, and Ahmad K Al Hwaitat. Unveiling the dark side of chatgpt: Exploring cyberattacks and enhancing user awareness. *Information*, 15(1):27, 2024.

- [7] Zubair Ali. Owasp lists 10 most critical large language model vulnerabilities. *CSO Online*, 2024. Accessed: 2024-10-11.
- [8] Muhammad Arsal, Bilal Saleem, Sommia Jalil, Muhammad Ali, Maila Zahra, Ayaz Ur Rehman, and Zia Muhammad. Emerging cybersecurity and privacy threats of chatgpt, gemini, and copilot: Current trends, challenges, and future directions. *Preprints*, October 2024.
- [9] Maurizio Atzori, Eleonora Calò, Loredana Caruccio, Stefano Cirillo, Giuseppe Polese, and Giandomenico Solimando. Evaluating password strength based on information spread on social networks: A combined approach relying on data reconstruction and generative models. *Online Social Networks and Media*, 42:100278, 2024.
- [10] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.
- [11] Minhaz Chowdhury, Nafiz Rifat, Shadman Latif, Mostofa Ahsan, Md Saifur Rahman, and Rahul Gomes. Chatgpt: The curious case of attack vectors' supply chain management improvement. In *2023 IEEE International Conference on Electro Information Technology (eIT)*, pages 499–504. IEEE, 2023.
- [12] Oleksandr Chybiskov. Hallucinations in llms: What you need to know before integration, 2023. Accessed: October 11, 2024.
- [13] Codecademy. Detecting hallucinations in generative ai, 2024. Accessed: 2024-12-05.
- [14] Codeium. Codeium security and privacy policy, 2024. Accessed: 2024-12-12.
- [15] GitHub Community. Github copilot - confusion around free access, 2023. Accessed: 2024-10-21.
- [16] Confident AI. The comprehensive guide to llm security, 2024. Accessed: 2024-10-10.
- [17] CyberNews. Black hat: Microsoft copilot data leak raises security concerns, 2024. Accessed: 2024-10-18.
- [18] Erik Derner and Kristina Batistič. Beyond the safeguards: exploring the security risks of chatgpt. *arXiv preprint arXiv:2305.08005*, 2023.
- [19] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems*, 36, 2024.
- [20] Plain English. Stop using cursor ai for code: The hidden dangers no one is talking about. *AI Plain English*, 2024. Accessed: 2024-12-10.
- [21] Gaper.io. Chatgpt data breach: What happened and how to protect yourself, 2024. ChatGPT utilizes an open-source library, called Redis, to access sensitive user data. The hackers took advantage of this vulnerability and gained access to chat histories and, in some cases, user payment information.
- [22] GitGuardian. Yes, github copilot can leak secrets, 2024. Accessed: 2024-10-18.
- [23] John Heibel and Daniel Lowd. Mapping your model: Assessing the impact of adversarial attacks on llm-based programming assistants. *arXiv preprint arXiv:2407.11072*, 2024.
- [24] Tonja Jacobi and Matthew Sag. We are the ai problem. *Emory Law Journal Online*, 74:1, 2024.
- [25] Andreas Jungherr. Using chatgpt and other large language model (llm) applications for academic paper assignments, 2023.
- [26] Arya Kavian, Mohammad Mehdi Pourhashem Kallehbasti, Sajjad Kazemi, Ehsan Firouzi, and Mohammad Ghafari. Llm security guard for code. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 600–603, 2024.
- [27] khaismile1997. Concerns about privacy mode and data storage. <https://forum.cursor.com/t/concerns-about-privacy-mode-and-data-storage/5418>, 2024. Accessed: 2024-12-10.
- [28] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruiyang Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.
- [29] LLM Security. Llm security, 2024. Accessed: 2024-10-10.
- [30] Microsoft. Privacy statement, 2024. Accessed: 2024-10-21.
- [31] Abhika Mishra, Akari Asai, Vidhisha Balachandran, Yizhong Wang, Graham Neubig, Yulia Tsvetkov, and Hannaneh Hajishirzi. Fine-grained hallucination detection and editing for language models. *arXiv preprint arXiv:2401.06855*, 2024.
- [32] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513*, 2024.

- [33] Zahra Mousavi, Chadni Islam, Kristen Moore, Alsharif Abuadbba, and M Ali Babar. An investigation into misuse of java security apis by large language models. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1299–1315, 2024.
- [34] Nexla. Ai infrastructure: Addressing large language model hallucinations. *Nexla*, 2024. Accessed: 2024-10-11.
- [35] University of Central Arkansas. Chat gpt: What is it?, 2024. Accessed: 2024-12-03.
- [36] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. Poisoned chatgpt finds work for idle hands: Exploring developers’ coding practices with insecure suggestions from poisoned ai models. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1141–1159. IEEE, 2024.
- [37] User on Cursor Forum. Ai hallucinations and code apply issues, 2024. Accessed: 2024-12-10.
- [38] OpenAI. Chatgpt capabilities overview, 2024. Accessed: 2024-12-03.
- [39] Oscar Oviedo-Trespalacios, Amy E Peden, Thomas Cole-Hunter, Arianna Costantini, Milad Haghani, JE Rod, Sage Kelly, Helma Torkamaan, Amina Tariq, James David Albert Newton, et al. The risks of using chatgpt to obtain common safety-related information and advice. *Safety science*, 167:106244, 2023.
- [40] PrivacyDesigner. Gdpr and ai risks for codeium. *PrivacyDesigner Blog*, 2024. Accessed: 2024-12-12.
- [41] Ahmed R Sadik, Antonello Ceravola, Frank Joublin, and Jibesh Patra. Analysis of chatgpt on source code. *arXiv preprint arXiv:2306.00597*, 2023.
- [42] Surf Security. Chatgpt via surf, 2024. Accessed: 2024-12-04.
- [43] SentinelOne. Chatgpt security risks: Threats and challenges of ai. <https://www.sentinelone.com/cybersecurity-101/data-and-ai/chatgpt-security-risks/>, 2023. Accessed: 2024-12-05.
- [44] Tushar Sharma. Llms for code: The potential, prospects, and problems. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, pages 373–374. IEEE, 2024.
- [45] Mohammed Latif Siddiq and Joanna CS Santos. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889*, 2023.
- [46] Spiceworks. Chatgpt leaks sensitive user data, openai suspects hack. *Spiceworks*, 2023. Accessed: 2024-12-03.
- [47] Acorn Team. Llm security. *Acorn.io*, 2024. Accessed: 2024-10-11.
- [48] Sangfor Technologies. Openai data breach and hidden risks for ai companies, 2023. Accessed: 2024-12-03.
- [49] Nearshore Technology. Conversations with ai: 7 features of chatgpt, 2023. Accessed: 2024-12-03.
- [50] TechRadar. Microsoft patches critical security bug in copilot studio that could have leaked private data, 2024. Accessed: 2024-10-18.
- [51] Surendrabikram Thapa and Surabhi Adhikari. Leveraging chatgpt-like large language models for alzheimer’s disease: Enhancing care, advancing research, and overcoming challenges. In *Smart Healthcare Systems*, pages 265–275. CRC Press, 2024.
- [52] Jixin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. Is your ai-generated code really secure? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395*, 2024.
- [53] Xiaodong Wu, Ran Duan, and Jianbing Ni. Unveiling security, privacy, and ethical concerns of chatgpt. *Journal of Information and Intelligence*, 2(2):102–115, 2024.
- [54] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340, 2022.
- [55] Jasmin Zernikow, Leonhard Grassow, Jan Gröschel, Philippe Henrion, Paul J Wetzel, and Sebastian Spethmann. Clinical application of large language models: Does chatgpt replace medical report formulation? an experience report. *Innere Medizin (Heidelberg, Germany)*, 2023.
- [56] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *arXiv preprint arXiv:2409.20550*, 2024.