The Journal of Supercomputing (2025) 81:776
https://doi.org/10.1007/s11227-025-07237-3

Evaluating ChatGPT's strengths and limitations for data race detection in parallel programming via prompt engineering

May Alsofyani[1] · **Liqiang Wang**[1]

Accepted: 24 March 2025

Abstract
Large Language Models have significantly advanced software engineering, enabling tasks like code comprehension and fault detection. However, their ability to detect complex bugs, such as data races in parallel programming, remains uncertain. Fault detection in parallel programming (Pthreads) requires a deep understanding of thread-based logic, as data races occur when threads access shared data concurrently without proper synchronization. This paper explores ChatGPT's potential in Pthreads fault detection by addressing three questions: **(1) Can ChatGPT effectively debug parallel programming threads? (2) How can dialogue assist with the detection of faults? (3) How can prompt engineering help to improve ChatGPT's fault detection performance?**. We examine advanced prompt engineering techniques, such as Zero-Shot, Few-Shot, Chain-of-Thought, and Retrieval-Augmented Generation prompts. Additionally, we introduce three hybrid prompting techniques to enhance performance, including Chain-of-Thought with Few-Shot Prompting, Retrieval-Augmented Generation with Few-Shot Prompting, and Prompt Chaining with Few-Shot Prompting, while evaluating ChatGPT's strengths and limitations for data race detection.

Keywords large language model · LLM4SE · Fault detection · Data race · Parallel programming

1 Introduction

Using large language models (LLMs), such as ChatGPT, can improve code comprehension by explaining code snippets, identifying faults or bugs, and suggesting how to fix them. The LLMs are a class of foundational models that have been trained on massive datasets to be able to understand and generate human-like text for a variety

✉ May Alsofyani
may.alsofyani@ucf.edu

Liqiang Wang
liqiang.wang@ucf.edu

[1] Department of Computer Science, University of Central Florida, Orlando, FL, USA

Published online: 23 April 2025

of applications. However, applications like ChatGPT represent specific implementations of LLMs, fine-tuned and optimized for interactive, task-specific use cases. This distinction is significant, as ChatGPT leverages the underlying LLM's capabilities and adapts them for practical applications in software engineering through prompt engineering and optimization techniques [1–5]. The studies explore ChatGPT's capabilities for repairing programs by identifying and correcting errors in source code to restore its intended functionality, understanding code, and generating code [6–11]. The findings suggest that LLMs outperform state-of-the-art program understanding and repair techniques. ChatGPT is a prominent example of how LLM applications can address specific challenges in the field.

A recent study investigated ChatGPT's effectiveness in program repair for deep learning models. It analyzed the impact of various prompt designs on its ability to perform debugging tasks, including fault detection, fault localization, and program repair. By incorporating buggy programs with more complex functionalities and dependencies, this study offers a more comprehensive evaluation of ChatGPT's performance in real-world debugging scenarios [11]. Another study highlights the advantages of using pre-trained language models for repair tasks, including complete function generation, correct code infilling, and single-line generation. Notably, these models demonstrate a strong ability to generalize across different programming languages and code patterns, enhancing their applicability in diverse software development contexts [9]. Furthermore, researchers found that automating the repair of Codex-generated code can significantly improve its scalability and reliability, especially in error detection and correction [10]. Codex [12] is a large language model developed by OpenAI that can understand and generate code, enabling tools like GitHub Copilot to assist developers with intelligent suggestions. Additional research indicates that LLMs, including ChatGPT and Codex, are more accurate and efficient in real-world program repair scenarios compared to traditional rule-based approaches [8]. Moreover, both ChatGPT and Codex have shown proficiency in code comprehension, syntax analysis, and code generation across a wide range of programming languages [6].

In this paper, we examine both the advantages and limitations of using large language models (LLMs) such as ChatGPT for debugging and fault detection in Pthread programs, focusing on ChatGPT due to its notable performance, accessibility, and impact among comparable LLMs. Our study aims explicitly to detect data race bugs in parallel programs for two main reasons. First, parallel programming languages differ significantly from sequential ones, as they are designed to execute multiple tasks simultaneously in various processing units, with built-in features to manage data sharing and synchronization. In parallel programs, data race bugs can lead to unpredictable behavior and errors that are challenging to reproduce and debug, as these bugs occur when two or more threads access shared data concurrently without proper synchronization, potentially causing data corruption and system crashes [13, 14]. Second, for ChatGPT to effectively debug Pthread programs, it must parse code syntactically and accurately identify and comprehend the complex interactions between multiple threads, which requires a deep understanding of concurrency issues.

To distinguish our research from previous studies [6–11], we selected Pthread programs as benchmarks due to their widespread use in parallel programming. Pthreads are extensively utilized in modern parallel programming, offering low-level con-

trol over thread management and synchronization. They are particularly well-suited for investigating complex concurrency issues such as data races and synchronization errors. To the best of our knowledge, no prior research has explored the use of large language models (LLMs) for analyzing Pthread-based programs. By focusing on Pthreads, we aim to fill this gap and provide novel insights into the application of AI in parallel programming.

Our research addresses three questions, beginning with: **RQ1. Can ChatGPT effectively debug Pthread programs?** This question evaluates ChatGPT's debugging performance for Pthread programs using a basic prompt for fault detection. As part of the evaluation, we measure the number of faulty programs detected and assess performance using accuracy, F1 score, recall, and precision metrics.

RQ2. How can dialogue assist with the detection of faults? Since ChatGPT is designed for interactive user engagement, its effectiveness in enhancing program fault detection in Pthread through dialogue-based interaction remains uncertain. This question aims to investigate whether providing hints about fault types (data races) and thread synchronization can further improve ChatGPT's performance in detecting faults.

RQ3: How can prompt engineering help to improve ChatGPT's fault detection performance? Studies have shown that prompt engineering can significantly improve the responses of LLMs [15–18]. However, determining the optimal prompts remains a challenge. We explore various prompt engineering strategies to address this, including Zero-Shot Learning, Few-Shot Learning, Chain-of-Thought, and Retrieval-Augmented Generation prompts. We also develop a guideline by creating multiple hybrid prompting templates. Based on our findings, we propose the most effective prompt template and apply it to our benchmark to demonstrate its effectiveness in enhancing fault detection.

We also summarize ChatGPT's strengths based on our findings and highlight key insights from the study. The contributions of this work are summarized as follows:

- Our study assesses ChatGPT's capability to detect data race bugs in Pthread programs by using both fixed and faulty programs. In the first benchmark, each faulty program contains a single bug, while in the second benchmark, faulty programs contain up to four bugs.
- We identify essential aspects for prompt design and propose optimized prompt templates that enhance the detection of bugs in parallel programs.
- We apply various prompt engineering techniques, including zero-shot prompting, Few-Shot prompting, Chain of Thought, and Retrieval-Augmented Generation, to improve ChatGPT's fault detection performance.
- We introduce three hybrid prompting techniques: Hybrid Prompting 1 (Chain-of-Thought with Few-Shot Prompting), Hybrid Prompting 2 (Retrieval-Augmented Generation with Few-Shot Prompting), and Hybrid Prompting 3 (Prompt Chaining with Few-Shot Prompting).
- We analyze ChatGPT's fault detection characteristics, finding that data race bugs are more effectively identified when Pthread synchronization scenarios are incorporated.
- We evaluate our study using accuracy, F1 score, recall, and precision metrics.

2 Related work

2.1 Large language model for software engineering (LLM4SE)

Large language models (LLMs) have demonstrated transformative potential across various domains, including software engineering, mathematics [19–21], high-performance computing (HPC) [22, 23], and natural language processing (NLP) [24, 25]. In software engineering, LLMs have been used for various tasks, including program repair [8, 9, 26–28], code summarization, and code generation [7, 29–32]. Recently, LLMs have gained a growing amount of attention in HPC research. For example, Chen et al. developed LM4HPC [22], which addresses HPC challenges by applying LLMs to code generation and performance optimization. Subsequently, HPC-GPT [23], a LLaMA-based model fine-tuned to HPC-specific question-answer pairs, was developed to enhance LLM applications in HPC, further expanding their capabilities in this specialized domain.

Beyond these tasks, LLMs have also been employed to tackle API-related challenges, such as detecting API misuse and answering API-related questions. For instance, Yangrui Yang et al. [33] developed APICKnow, a model for extracting API entities and their semantic relationships from texts based on large language models. Stack Overflow datasets have been used in experiments to prove APICKnow's effectiveness at achieving high accuracy without requiring large amounts of labeled data. Similarly, Huang et al. [34] developed a novel approach to resolve fully qualified names (FQNs) in partial code snippets by fine-tuning a code-masked language model (MLM). Studies like these highlight the capabilities of LLMs but also identify areas for improvement and provide insights into their practical use.

In the domain of automated program repair (APR), Jiang et al. [26] analyzed the performance of various language models on real-world software bugs. In their study, multiple models were compared, factors like bug types and code complexity were examined, and strengths and limitations were identified. As a result of their findings, language models demonstrate the potential for enhancing APR techniques while also highlighting potential areas for improvement. Ouyang et al. [35] introduced reinforcement learning with human feedback (RLHF) as a method of training language models to follow instructions more effectively. Similarly, Christiano et al. [36] proposed a method for training deep reinforcement learning agents that align actions with desired outcomes using human preferences. These advancements have encouraged researchers to apply these methods to code-related tasks, such as program repair [16] and code generation [37].

One notable study identifies key obstacles when handling multi-fault programs using existing repair approaches by examining the repair problem of multi-fault programs [38]. This study provides valuable insights into how to deal with complex repair scenarios beyond single-fault scenarios. Several tools have been developed for automated program repair for parallel programming, where errors are often more complicated. For instance, LLOR [39] offers significant capabilities for repairing parallel programs by addressing data race errors by introducing appropriate synchronization constructs. Meanwhile, GPURepair [40] extends automated program repair to GPU

kernels written in CUDA and OpenCL, focusing on resolving data races and barrier divergence errors.

A recent study evaluated ChatGPT's capabilities for repairing code for deep learning programs by evaluating various prompt designs to detect and resolve code issues [11]. Other research highlights the benefits of pre-trained language models for repairing programs, particularly their ability to generalize across languages and code patterns [9]. Moreover, LLMs such as ChatGPT and Codex outperform traditional rule-based methods in program repair [8] and demonstrate strong skills in code comprehension, syntax analysis, and code generation across a wide range of programming languages [6]. In addition to improving scalability and reliability, Codex detects and corrects errors during automated repair tasks.

In a few-shot learning procedure for code-related tasks, Nashid et al. [28] presented a retrieval-based approach to guide language models in repairing code and generating test assertions. Their method demonstrated significant performance gains, highlighting the value of retrieval-based prompt selection for improving few-shot learning. Similarly, Ahmed et al. [32] focused on project-specific code summarization, leveraging few-shot training with project-specific examples to fine-tune LLMs, leading to better relevance and quality in code summaries.

Based on this work, our research investigates the detection of more complex faulty programs, including data races in Pthread code. To better understand ChatGPT's capabilities to handle concurrency-related bugs, we evaluate alternative prompt templates and the strengths and limitations of its responses.

2.2 Parallel programming

By utilizing multicore processors, parallel programming (Pthreads) allows multiple threads to execute concurrently, improving program performance. While thread management can be very beneficial, it can also introduce challenges like concurrency bugs, particularly data races, which occur when multiple threads simultaneously synchronize their access to a single variable. To prevent unpredictability and system crashes, synchronization mechanisms like condition variables and mutex locks are implemented to ensure thread safety. Effective detection and repair methods are needed to address data race. Previous and recent research has proposed various detection and repair methods, such as static analysis, dynamic analysis, and runtime verification [41–43]. These techniques can be used to detect data races and, in some cases, repair them. Serebryany et al. [44] proposed a dynamic race detection method integrated with the LLVM compiler, using compile-time instrumentation to enhance ThreadSanitizer's capabilities. Similarly, another study [45] introduced a sound and partially complete static analysis technique for detecting data races in GPU programs, combining static analysis with domain-specific abstractions to address GPU concurrency challenges. Choi et al. [46] developed a precise method for detecting data races in multithreaded object-oriented programs by leveraging static and dynamic analysis techniques. RaceFixer [47], an automated tool, identifies and fixes data races by analyzing concurrency issues and recommending synchronization mechanisms such as mutex locks. Additionally, Shastri et al. [48] presented HMTRace, a hardware-assisted memory-tagging approach for

dynamic data race detection, utilizing memory-tagging hardware to improve scalability and efficiency.

While many techniques have been proposed for detecting and resolving data races, LLMs offer distinct advantages that make them invaluable in modern parallel programming. In recent years, research has significantly advanced the understanding of how large language models can enhance parallel programming [49–53]. OMPify [49], for example, uses a graph-based representation of source code in conjunction with a Transformer model to predict OpenMP pragmas and analyze loop parallelization. OpenMP (Open Multi-Processing) is an API that allows developers to write parallel code on multiple platforms with shared memory. Furthermore, OMPGPT [50], a domain-specific LLM, has been developed to assist in generating OpenMP pragmas to illustrate the potential of domain-specific LLMs for automating parallelization. ChatGPT and GitHub Copilot have also been evaluated in several studies for generating parallel code [52] and detecting concurrency problems in OpenMP [53]. Studies like these and others emphasize the importance of LLMs in parallel programming, particularly in providing intelligent code suggestions and automating complex tasks such as loop parallelization and paradigm generation. Their adaptability to diverse programming paradigms and capacity to learn from extensive code repositories enable them to evolve continuously and remain effective in rapidly changing technological environments. These capabilities, combined with their accessibility and ability to integrate detection, repair, and explanation tasks, position LLMs as indispensable tools for advancing parallel programming and educating future developers.

3 Study design

3.1 Benchmark

Pthreads and other parallel programming languages enable multiple threads to run simultaneously across various processing units. These languages include built-in features to manage data sharing and synchronization, ensuring efficient and safe parallel execution. A data race bug occurs when two or more threads access shared variables concurrently without proper synchronization, leading to data corruption or unpredictable behavior. To prevent data corruption, mutexes are commonly used as synchronization mechanisms to protect shared variables. A mutex allows only one thread to access a shared resource at a time. The `pthread_mutex_lock()` function locks a mutex object, granting the calling thread exclusive ownership until it is released using `pthread_mutex_unlock()`. If another thread attempts to lock an already-locked mutex, it must wait until the mutex becomes available, ensuring safe and controlled access to shared variables. For our study, we extracted 88 source code files from open-source projects on GitHub, focusing on specific `mutex_lock` and `mutex_unlock` synchronization patterns to analyze these essential concurrency practices. We divided our benchmark into two sets, which will be discussed in the fol-

lowing section. Each program in the benchmark contains approximately 50 to 250 lines of code.[1]

3.2 Creating data race bugs

The benchmark suite utilized in this study encompasses both faulty and fixed Pthread programs. The faulty programs were generated by omitting lines of code associated with synchronization mechanisms, specifically those invoking `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

- The first benchmark comprises 22 fixed source code files that implement `pthread_mutex_lock()` and `pthread_mutex_unlock()`, alongside 22 faulty files where these synchronization mechanisms were removed. Each faulty file in this benchmark contains only one type of data race related explicitly to mutex synchronization patterns. To analyze the impact of dataset distribution on fault detection, we evaluated our first benchmark with two settings: a balanced dataset, which includes an equal number of faulty and fixed files, and an imbalanced dataset, where faulty and fixed files are unevenly distributed. This approach allows us to assess how variations in dataset composition influence ChatGPT's performance in detecting data races.
- The second benchmark also contains 22 fixed source code files with `pthread_mutex_lock()` and `pthread_mutex_unlock()`, along with 22 faulty files where these functions were removed. However, unlike the first benchmark, each faulty file in this set contains multiple data races, ranging from two to four types, resulting from mutex synchronization issues. By structuring our benchmarks this way, we aimed to systematically evaluate the effectiveness of different prompting techniques in detecting single and multiple data race conditions. This design comprehensively assesses ChatGPT's capabilities in handling concurrency-related bugs.

3.3 ChatGPT setup

We developed an experimental pipeline using the ChatGPT API, explicitly leveraging the GPT-4-turbo model from OpenAI. To address ChatGPT's non-deterministic behavior, we performed three independent runs for each query, starting a fresh conversation with the API each time. Throughout the process, we documented all prompts and their corresponding responses.

3.4 Evaluation metrics

To account for ChatGPT's stochastic nature, we followed a methodology inspired by prior research [11]. Specifically, we repeated each fault detection request three times, initiating a new conversation with the API for each trial. If the majority of responses (i.e., at least 2 out of 3) correctly identify a fault, we record the result as '1'

[1] The benchmark can be found at https://github.com/MayAlsofyani/Pthread-Benchmark.

(indicating a data race bug); otherwise, we mark it as '0' (indicating a bug-free file). We then applied evaluation metrics commonly used in software engineering research, calculating accuracy, F1 score, precision, and recall for each prompt. Finally, we take the average of these metrics across the three trials to provide a comprehensive assessment of fault detection performance.

4 Methodology

4.1 Pthread program debugging

RQ1. Can ChatGPT effectively debug Pthread programs?

This section aims to evaluate ChatGPT's capability to debug parallel programs using our benchmarks. As shown in Listing 1, we adapted a basic prompt template from previous research [16] and used this prompt as the baseline for our work. This template includes a CODE section where each fixed and faulty program from the benchmark is inserted. The overall results for RQ1 are presented in Tables 1 and 2, with ChatGPT's performance using a basic prompt labeled as (Basic Prompt).

```
Prompt Template:
Is the following program buggy? If so, please respond with '1';
     otherwise, respond with '0'.

   {CODE}

Final Response:
   "1" or "0"
```

Listing 1 Prompt Template for Basic Prompt (Baseline)

We followed ChatGPT's variability evaluation protocol, as discussed in Section 3.4. After three runs, we observed that ChatGPT labeled all files as faulty, assigning a value of '1' to each of the files in our benchmark. After obtaining the results, we asked ChatGPT to provide reasons for labeling a program as faulty or, if it was marked as bug-free, to respond with '0' and provide the reasoning for that labeling. There were, however, a number of times when ChatGPT's explanations were unsatisfactory. Due to the complexity of Pthread programs, ChatGPT often responds with more general or vague descriptions, as it lacks the information needed for precise identification. Here are some examples of ChatGPT's explanations and responses.

Response: "The program provided has several potential issues that could lead to faults during execution."
Response: "The program has a potential fault related to the use of the mutex."
Response: "The program is faulty due to: Dynamic Initialization."
Response: "The program is faulty due to: Termination."

The complexity of Pthread programs presents significant challenges for ChatGPT, often resulting in responses that are broad or vague. This occurs because Pthread

programs involve intricate concurrency issues-such as data races, deadlocks, and synchronization-that demand a deep understanding of thread interactions. ChatGPT lacks the ability to interpret these dynamic runtime behaviors directly. Although it is trained on general information and debugging concepts, ChatGPT's knowledge may not extend to specialized areas like Pthreads and synchronization primitives. Without prompts that include specific details related to parallel programming, ChatGPT tends to default to general descriptions, as it lacks the context needed to identify particular issues accurately.

More targeted prompts can be utilized to improve ChatGPT's performance in analyzing threads. For example, incorporating code snippets as examples can help ChatGPT more accurately identify data race bugs and distinguish between faulty and fixed programs. Including more detailed information, such as specifying fault types like "data race in mutex usage and synchronization primitives," can further guide its responses. Additionally, using clear and concise language in the prompts can enhance the quality of ChatGPT's answers. Incorporating specific scenarios within the prompts can help direct the model toward the desired outcome. Breaking down complex issues into smaller, focused questions can also aid in obtaining more precise and informative answers.

Finding 1: Using the Basic Prompt as the initial template in this study, ChatGPT labeled all files as faulty, identifying bugs in each and assigning a value of 1 to all files in our benchmarks. *The results indicate that more targeted prompts can be used, and additional information – including terms such as "faulty type" or "synchronization primitives" – is needed to ensure ChatGPT's responses are as accurate as possible.*

4.2 Impact of dialogue

RQ2. How can dialogue assist with the detection of faults?

In RQ1, ChatGPT's performance was less than satisfactory, motivating us to draft and design an improved zero-shot prompt. The revised prompt included keywords that guided ChatGPT toward the desired response format. Furthermore, clarifying the context and using concise language helped narrow the focus with relevant keywords like "data race fault type,""synchronization problem," and "parallel programming." As part of our efforts to enhance ChatGPT's fault detection performance, we employed the prompt engineering technique: Zero-Shot Learning Prompt.

4.2.1 Zero-shot learning prompt

This approach involves giving ChatGPT the task of classifying faulty programs from fixed ones without prior examples. Listing 2: In designing our first zero-shot prompt, we included keywords such as "parallel programming codes," "data race", and "synchronization problem" and instructed ChatGPT to examine all codes. Due

to ChatGPT's inherent variability, we adhered to our experimental setup using the Zero-Shot Learning Prompt. We calculated our metrics, as shown in Table 1 for the first benchmark (one bug per file) and Table 2 for the second benchmark (up to four bugs per file).

```
Prompt Template:

Examine the following parallel programming code and determine
    if there is a synchronization problem, such as a Data Race.
     If a problem exists, respond with "1"; if none, respond
    with "0"

   {CODE}

Final Response:
   "1" or "0"
```

Listing 2 Prompt Template for Zero-Shot Prompt Template

Using specific keywords, we tested ChatGPT's performance in various combinations and observed improvements in accuracy. ChatGPT performed better when it used precise keywords like "parallel programming" instead of "Pthread" and "synchronization problem" instead of the broader term "synchronization issue." As opposed to general terms like "issue," ChatGPT provided more accurate responses when prompted to identify synchronization problems, enabling it to detect faults more accurately when prompts were tailored to specific issues. Furthermore, ChatGPT's fault detection accuracy increased by approximately 61% when using the Zero-Shot Learning Prompt in both the first and second benchmarks.

Finding 2: Using keywords such as "parallel programming", "synchronization problem", and "data race" resulted in more accurate responses. *Adding more examples to prompts, along with contextual information, can further enhance ChatGPT's accuracy.*

4.3 Impact prompt engineering

RQ3. How can prompt engineering help to improve ChatGPT's fault detection performance?

In RQ2, ChatGPT's performance was somewhat satisfactory, which led us to design a more effective prompt. The revised prompts incorporated specific keywords, such as from RQ2, to narrow the focus, and we also planned to include examples to guide ChatGPT toward the desired response format.

A number of advanced prompting techniques were used to enhance ChatGPT's fault detection performance, including a Few-Shot Learning Prompt, a Chain-of-Thought Prompt, and a Retrieval-Augmented Generation (RAG) Prompt. Additionally, we introduced three hybrid prompting techniques: Hybrid Prompting 1 (Chain-of-Thought with Few-Shot Prompting), Hybrid Prompting 2 (Retrieval-Augmented

Generation with Few-Shot Prompting), and Hybrid Prompting 3 (Prompt Chaining with Few-Shot Prompting).

4.3.1 Few-shot learning prompt

```
Prompt Template:

Examine the following parallel programming code and determine
    if there is a synchronization problem, such as a Data Race.
     If a problem exists, respond with "1"; if none, respond
    with "0".

   Example 1:
   Code:
       pthread_mutex_lock(&lock);
       pthread_mutex_unlock(&lock);
   Response: 0

   Example 2:
   code:
       pthread_mutex_lock(&lock);
   Response: 1

   Example 3:
   code:
   Response: 1:

   {CODE}

Final Response:
   "1" or "0"
```

Listing 3 Prompt Template for Few-Shot Learning Prompt

This approach involved providing ChatGPT with examples of synchronization mechanisms in Pthread programs, as illustrated in Listing 3, to help it recognize relevant mutex statements, specifically `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The provided examples covered both correct and incorrect usage of mutex statements. Proper synchronization examples demonstrated how mutexes should be used to lock and unlock the same shared variable, exposing ChatGPT to fundamental synchronization patterns. Additionally, we included incorrect usage scenarios, such as missing mutex operations, to help the model identify and flag potential synchronization issues more effectively. In designing the few-shot prompt, we built upon our previous zero-shot prompt, maintaining its key structure and keywords while adding relevant examples. As shown in Table 1, using the few-shot learning prompt, we achieved 75% accuracy in fault detection in our first benchmark, where each file contains one bug, and 71% accuracy in the second benchmark, where each file contains multiple bugs.

Incorporating examples into prompts significantly enhanced ChatGPT's ability to recognize and understand synchronization mechanisms, leading to more accurate responses. By providing context through examples, the model developed a better grasp of mutex usage patterns, including `pthread_mutex_lock()` and

`pthread_mutex_unlock()`, and their role in ensuring proper synchronization. This approach notably improved ChatGPT's learning efficiency and fault detection performance. The impact of including examples was evident in the accuracy improvement achieved with the few-shot prompt compared to the zero-shot approach, reinforcing the importance of example-driven learning in enhancing the model's ability to understand concurrency issues and make more precise decisions.

4.3.2 Chain-of-thought prompt

While few-shot learning improves ChatGPT's performance by providing specific examples to guide its understanding, Chain-of-Thought (CoT) prompting takes a different approach by breaking down the reasoning process into detailed, logical steps. This method encourages the model to articulate intermediate steps in its reasoning rather than arriving directly at a conclusion [54]. In listing 4, we designed our steps as follows:

1. Check if the code involves shared variables accessed by multiple threads.
2. Identify race conditions by checking if shared variables are modified without proper synchronization.
3. If synchronization `pthread_mutex_lock()` and `pthread_mutex_unlock()` is correctly used around critical sections, respond with '0'.
4. If no synchronization or improper synchronization is detected, respond with '1'.

```
Prompt Template:

You are an expert. Your task is to analyze the following
    parallel programming code for synchronization problem,
    specifically looking for data race.If problem is present,
    respond with '1' or '0' if none. Follow the steps below:

   1. Check if the code involves shared variables accessed by
       multiple threads.
   2. Look for race conditions: Are shared variables modified
       without proper synchronization (e.g., without
       pthread_mutex_lock/unlock)?
   3. If synchronization (pthread_mutex_lock/unlock) is used
       correctly around critical sections, respond with "0".
   4. If no synchronization or improper synchronization is
       detected, respond with "1".

   {CODE}

Finla Response:
   "1" or "0"
```

Listing 4 Prompt Template for Chain-of-Thought Prompt

By evaluating our first benchmark, which contains one bug per file, ChatGPT achieved an accuracy of 65% when utilizing the Chain-of-Thought (CoT) prompting approach with the outlined reasoning steps. The accuracy of our second benchmark, which includes multiple bugs per file, dropped to 60%. This limited performance can be attributed to several factors. One key challenge is the inherent complexity of recognizing and accurately identifying synchronization issues in multithreaded programs.

These issues often require a deeper contextual understanding than what the structured CoT steps alone can provide. The second challenge is the variability in how synchronization problems manifest across different code snippets, which can introduce ambiguity, making it difficult for the model to generalize effectively.

Given these challenges, we observed that incorporating examples significantly enhanced ChatGPT's performance. Few-shot prompting, which provides the model with concrete examples of buggy and non-buggy scenarios, improved its accuracy by offering more straightforward guidance. By seeing explicit cases, ChatGPT was better able to recognize patterns, reduce misclassifications, and enhance its overall bug-detection capabilities.

Finding 3: Providing ChatGPT with examples helped it to better understand the prompt, resulting in better performance. *This demonstrates that the key to success is to provide ChatGPT with context and clarity.*

4.3.3 Hybrid prompting 1: chain-of-thought with few-shot prompting

To enhance the Chain-of-Thought approach, we redesigned the Chain-of-Thought prompt to include example-based steps. This combined prompt, named "Hybrid Prompting 1: Chain-of-Thought with Few-Shot Prompting," integrates a step-by-step thought process with few-shot learning examples.

```
Prompt Template:

You are an expert. Your task is to analyze the following
    parallel programming code for synchronization problem,
    specifically looking for data race. If problem is present,
    respond with "1" or "0" if none.
   Follow the steps below:

   1. Check for shared variables and critical sections.
   2. If both "pthread_mutex_lock" and "pthread_mutex_unlock"
       are used correctly, respond with "0" (no data race).
   3. If either is missing or improperly used, respond with "1"
        (data race present).

   Here are some examples:
       Example 1:
         Code: pthread_mutex_lock(&lock); shared_var++;
               pthread_mutex_unlock(&lock);
       Response: 0

       Example 2:
         Code: shared_var++;
       Response: 1

   Now, analyze this code:  {CODE}

Final Response:
   "1" or "0"
```

Listing 5 Prompt Template for Hybrid Prompting: Chain-of-Thought with Few-Shot Prompting

Hybrid Prompting 1 begins with a step-by-step breakdown of the problem, where each step is clearly defined to guide ChatGPT's reasoning process. In the first step, shared variables are identified, followed by the second and third steps, which verify synchronization mechanisms, as shown in Listing 5. After these steps, we apply examples from the few-shot prompt, incorporating them in the final step to illustrate the expected results and further clarify the task.

By following this structured approach, ChatGPT leverages CoT reasoning and few-shot examples to enhance its understanding and improve accuracy. This hybrid method significantly boosted performance, achieving 75% accuracy on the first benchmark and 78% on the second, as shown in Tables 1 and 2. The combination of explicit reasoning steps and concrete examples effectively mitigated ambiguity and improved ChatGPT's ability to detect synchronization issues in multithreaded programs.

4.3.4 Hybrid Prompting 2: Retrieval-Augmented Generation with Few-Shot Prompting

The Retrieval-Augmented Generation (RAG) prompt is a technique that enhances language model performance by combining AI's generation capabilities with the retrieval of relevant information from external sources [55, 56]. This approach allows the model to access and incorporate specific or up-to-date data, improving its ability to generate accurate and contextually rich responses. By integrating retrieval mechanisms, the model can provide more informed answers, particularly for tasks that require detailed knowledge. We chose these sources [57, 58] to retrieve and incorporate specific, up-to-date data based on their inclusion of parallel programming code examples. These examples provided the model with rich information about synchronization mechanisms, which it used to improve its predictions. This approach enabled us to develop a more accurate and contextually rich model, particularly for fault detection and analysis.

Our third finding revealed that providing ChatGPT with examples improved its understanding of prompts and enhanced its performance. Building on this insight, we developed a hybrid prompt named "Hybrid Prompting 2," which combines Retrieval-Augmented Generation (RAG) with Few-Shot Prompting. This approach leverages the strengths of both retrieval and few-shot techniques to boost model performance. By incorporating retrieval, the model gains access to specific external information that enhances its contextual understanding. At the same time, few-shot prompting provides concrete examples that help guide the model in generating more precise and coherent responses. Combining these techniques, Hybrid Prompting 2 ensures that ChatGPT benefits from structured knowledge retrieval and practical learning from examples.

By adopting this hybrid approach, we achieved 72.0% accuracy on our first benchmark, which contains one bug per file-a substantial improvement over the 53.7% accuracy obtained using RAG alone, as shown in Table 1. Similarly, for our second benchmark, which includes multiple bugs per file, RAG alone resulted in 60% accuracy, whereas Hybrid Prompting 2 increased performance to 69.8%, as demonstrated in Table 2. These results highlight the effectiveness of combining retrieval-based augmentation with few-shot examples, demonstrating that Hybrid Prompting 2 provides

a more robust and adaptable approach to improving ChatGPT's ability to detect data races in Pthreads programs.

4.3.5 Hybrid prompting 3: prompt chaining with few-shot prompting

Based on our observations on our Hybrid Prompting 1 and 2 and our assumption, providing ChatGPT with examples improved its understanding of the task. We then implemented the Prompt Chaining method for fault detection in parallel programming. This approach involves breaking down a complex task into small prompts, feeding them sequentially to ChatGPT, and chaining the outputs to form a comprehensive final response. As shown in Listing 6, our Prompt Chaining setup includes small prompts fed sequentially to ChatGPT.

Prompt Chaining provides a structured way to handle the complexities of parallel programming by breaking down intricate problems into manageable sub-tasks. This sequential approach allows ChatGPT to process and understand complex tasks through incremental steps, where each step builds upon the previous one. Moreover, this approach enhances fault detection in parallel programming by allowing ChatGPT to focus on distinct aspects of data race detection before making a final decision.

We developed our Hybrid Prompting 3, which integrates Prompt Chaining with Few-Shot Prompting, as shown in Listing 6. Hybrid Prompting 3 consists of three prompts: the first prompt identifies all shared variables in the code, the second prompt (a Few-Shot Prompt) examines synchronization around shared variables using few-shot examples, which include proper and improper synchronization patterns, and the final prompt employs the findings from the previous steps to determine whether synchronization is correctly implemented or if data races exist.

By combining Prompt Chaining with Few-Shot Prompting, Hybrid Prompting 3 achieved an accuracy of 77% and 75% in both benchmarks, respectively, as shown in Tables 1 and 2. This result represents the highest accuracy among all tested prompting methods, as demonstrated in Table 1, particularly on our first benchmark, which contains one bug per file. The significant improvement in accuracy highlights the effectiveness of Hybrid Prompting 3 in enhancing ChatGPT's ability to detect synchronization issues in more accurate and coherent responses.

```
1  Prompt Template 1:
2  You are an expert. Your task is to analyze the following
       parallel programming code for synchronization problems,
       such as a Data Race. If a problem exists, respond with '1';
        if none, respond with '0'.
3
4  Check all shared variables accessed by multiple threads.
5     {CODE}
6     Response 1: {shared_variables} List all shared variables.
7
8  ------------------------------------------------------------
9  Prompt Template 2:
10 Are shared variables modified without proper synchronization (
       no pthread_mutex_lock/unlock)?
11
12        Here are some examples:
13             Example 1:
```

```
                Code:
                    pthread_mutex_lock(&lock);
                    Shared Variables
                    pthread_mutex_unlock(&lock);
                Response: 0

                Example 2:
                code:
                    pthread_mutex_lock(&lock);
                    Shared Variables
                Response: 1

                Example 3:
                code:
                    Shared Variables
                Response: 1

    Result from the previous step: {shared_variables}
    Response 2: {synchronization_check}

------------------------------------------------------------
Prompt Template 3:
1. If synchronization (pthread_mutex_lock/unlock) is used
    correctly around Shared Variables, respond with "0".
2. If no synchronization or improper synchronization is
    detected, respond with "1".

    Result from the previous step: {synchronization_check}

Final Response:
    "1" or "0"
```

Listing 6 Prompt Template for Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting

5 Discussion

To evaluate ChatGPT's performance, we implemented various prompt engineering techniques and summarized the results in this section. With the Basic Prompt, we observed that ChatGPT struggled to differentiate between faulty and fixed files, incorrectly labeling all files as faulty across both benchmarks-one containing a single bug per faulty file and the other with two or more bugs per file. This prompt was adapted from previous research [16] and served as the baseline for our study.

As shown in Tables 1 and 2, we evaluated the effectiveness of each prompt by measuring its accuracy in fault detection. Given ChatGPT's inherent variability, we repeated each fault detection request three times per faulty program, initiating a new conversation for each trial. We then calculated accuracy, F1 score, precision, and recall for each prompt, averaging these metrics across the three trials to ensure a comprehensive and reliable evaluation of fault detection performance.

By including specific keywords such as "parallel programming codes," "data race," and "synchronization problem" to guide ChatGPT in analyzing the code, we achieved an accuracy improvement of approximately 61% with the Zero-Shot Learning Prompt

in both our first benchmark (one bug per file) and second benchmark (multiple bugs per file).

Furthermore, our findings demonstrate that example-based prompts, such as the Few-Shot Learning Prompt, significantly enhance ChatGPT's accuracy, particularly for complex issues like data race detection in parallel programs. We employed our example-based prompt with scenarios where mutex statements were correctly applied, demonstrating proper locking and unlocking of the shared variable. These structured examples exposed ChatGPT to fundamental synchronization patterns, enabling it to generalize these rules for analyzing new code more effectively. Additionally, we incorporated examples of improper usage, such as missing or misused mutex operations, to help the model recognize and flag potential synchronization issues with greater precision. As a result of including these examples in the Few-Shot Prompt, we achieved a notable accuracy of 75% in the first benchmark and 71% in the second benchmark, as shown in Tables 1 and 2, emphasizing the critical role of examples in guiding ChatGPT toward more accurate and reliable fault detection.

As shown in Table 1, we evaluated our first benchmark (one bug per file) across all prompt designs and observed significant performance improvements with our hybrid approaches. Hybrid Prompting 1, which integrates Chain-of-Thought (CoT) with Few-Shot Prompting, increased accuracy to 75.0%, compared to 60% achieved with CoT alone. Similarly, Hybrid Prompting 2, which combines Retrieval-Augmented Generation (RAG) with Few-Shot Prompting, improved accuracy to 72.0%, compared to 53.7% achieved with RAG alone.

In contrast, Table 2 presents the results from the second benchmark (multiple bugs per file). Hybrid Prompting 1, which combines Chain-of-Thought (CoT) with Few-Shot Prompting, achieved the highest accuracy of 78.0%, demonstrating its effectiveness in handling more complex fault detection scenarios. CoT and RAG alone attained an accuracy of 60%, while Hybrid Prompting 2, which integrates RAG with Few-Shot Prompting, improved accuracy to 69.8%.

As a result, Hybrid Prompting 3 (Prompt Chaining with Few-Shot Prompting) achieved the highest accuracy of 77.7% in the first benchmark, Table 1, and the second-highest accuracy of 75.3% in the second benchmark, Table 2. The slightly lower accuracy in the second benchmark can be attributed to the increased complexity of the files, each containing up to four bugs. Each hybrid technique leverages the strengths of few-shot learning to address specific challenges in parallel programming, enhancing ChatGPT's fault detection capabilities.

Additionally, we compared ChatGPT's performance with Cursor [59], another large language model, as a baseline. We utilized Cursor Chat with a simple query, without providing examples: *'Examine all code to identify synchronization issues, such as a data race. If a problem is detected, output 1; otherwise, output 0.'*. Using this approach, Cursor achieved an accuracy of 73.0% in the first benchmark, one bug per file, and 62.0% in the second benchmark, multiple bugs per file. We then refined the query by incorporating examples for Cursor, which improved accuracy to 76.0% in the second benchmark, while the accuracy for the first benchmark remained unchanged at 62.0%. This discrepancy is due to differences in code snippets and the location of data races in the second benchmark. Although Cursor demonstrated strong performance in both benchmarks, it remained less effective than our hybrid prompting techniques in

Table 1 Performance Comparison of Prompting Techniques for Our First Benchmark (One Bug per File) with Basic Prompt and Cursor as Baselines

Prompts	Accuracy	F1	Recall	Precision
Basic Prompt (Baseline)	%50.0	%66.0	%100	%50.0
Cursor simple query (Baseline Model)	%73.0	%65.0	%48.0	%100
Cursor with examples (Baseline Model)	%61.0	%37.0	%23.0	%100
Zero-Shot Learning Prompt	%61.0	%66.6	%77.2	%58.6
Few-Shot Learning Prompt	%75.0	%74.4	%72.7	%76.1
Chain-of-Thought Prompt	%65.0	%68.0	%75.0	%62.0
Retrieval-Augmented Generation Prompt	%53.7	%62.2	%78.7	%51.8
Hybrid Prompting 1: Chain-of-Thought with Few-Shot Prompting	%75.0	%74.4	%72.7	%76.1
Hybrid Prompting 2: Retrieval-Augmented Generation with Few-Shot Prompting	%72.0	%72.6	%68.1	%73.6
Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting	**%77.7**	**%76.8**	**%72.7**	**%80.0**

Table 2 Performance comparison of prompting techniques for our second benchmark (multiple bugs per file) with basic prompt and cursor as baselines

Prompts	Accuracy	F1	Recall	Precision
Basic Prompt (Baseline)	%50.0	%66.6	%100	%50.0
Cursor simple query (Baseline Model)	%62.0	%58.0	%52.0	%65.0
Cursor with examples (Baseline Model)	%76.0	%67.0	%50.0	%100
Zero-Shot Learning Prompt	%61.0	%72.0	%100	%56.2
Few-Shot Learning Prompt	%71.4	%77.4	%98.4	%63.9
Chain-of-Thought Prompt	%60.3	%71.6	%100	%55.8
Retrieval-Augmented Generation Prompt	%60.3	%71.3	%98.4	%55.9
Hybrid Prompting 1: Chain-of-Thought with Few-Shot Prompting	78.5%	82.3%	100%	70.0%
Hybrid Prompting 2: Retrieval-Augmented Generation with Few-Shot Prompting	%69.8	%75.0	%90.4	%64.2
Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting	%75.3	%78.9	%92.0	%69.0

handling data race bugs. Nevertheless, its accuracy is comparable to advanced hybrid prompting techniques in ChatGPT, making it a valuable baseline for evaluating more sophisticated fault detection methods.

In this study, we aim to use an equal number of faulty and fixed files in each benchmark. Additionally, we experimented using an imbalanced dataset in the first benchmark to evaluate the effectiveness of our prompt engineering techniques. This experiment allowed us to assess how well ChatGPT performs when exposed to datasets with varying distributions of faulty and fixed code. Our findings show that Hybrid Prompting 3 achieved the highest accuracy in both settings: 77.7% in the balanced dataset and 77.5% in the imbalanced dataset. Similarly, Hybrid Prompting 1, which integrates Chain-of-Thought (CoT) with Few-Shot Prompting, consistently achieved 75.0% accuracy in balanced and imbalanced scenarios. However, the accuracy of the Few-Shot Learning Prompt dropped to 70.0%, while CoT and Hybrid Prompting 2 both declined to 60.0% in the imbalanced dataset. These results highlight the importance of using balanced datasets to ensure consistent and reliable fault detection performance across different prompting strategies.

While our study primarily focused on Pthreads, the synchronization challenges addressed-particularly the correct and incorrect use of mutexes-are commonly encountered in other shared-memory parallel programming models. Given that mutual exclusion and critical section management are foundational across these paradigms, the prompt engineering strategies and insights developed in this work are likely transferable beyond the scope of Pthreads. Notably, example-based prompting significantly improves ChatGPT's ability to recognize and reason about synchronization mechanisms. Furthermore, decomposing complex tasks into smaller subtasks or guiding the model through a step-by-step reasoning process within prompts is essential for improving its analytical performance, as it provides structured and context-rich scenarios for interpretation.

6 Threats of validity

Our study is susceptible to several potential threats to validity. The primary measure is how well ChatGPT's predictions identify concurrency bugs in Pthread programs. ChatGPT's probabilistic model, however, introduces inherent variability, as its responses may be influenced by latent factors beyond our control. To mitigate this problem, we designed prompts to yield consistent responses and validated their structure over multiple iterations. It is also essential to recognize that ChatGPT's randomness significantly threatens its validity. As a result, we ran each experimental setting three times for RQ2 and RQ3, ensuring that the variability was accounted for and reliability was increased. In addition, since ChatGPT is not deterministic, slight prompt variations or session differences could affect its results. We minimized these risks by using standardized prompts and consistent testing conditions. We also need to consider the accuracy of our manually labeled dataset, which serves as ground truth; errors in labeling could affect ChatGPT's prediction accuracy. Our study focuses specifically on mutex lock synchronization using the Pthreads benchmarks. While this provides a controlled and representative environment for testing ChatGPT's capabilities, it does

not encompass atomic operations, condition variables, or thread pools, among other concurrency issues. As a result, the findings may not be generalizable to different synchronization methods or concurrency paradigms. To enhance external validity, our prompts and ChatGPT's responses are available upon request by contacting the corresponding author. Additionally, an example of the prompt used in our study can be found in Appendix A, and our benchmarks are publicly available on GitHub. With these threats acknowledged and mitigating strategies implemented, we aim to provide a robust foundation for interpreting our results and highlight areas for future research to address the limitations of prompt-based concurrency analysis.

7 Conclusion

This paper evaluates ChatGPT's ability to detect data race bugs in mutex lock and mutex unlock synchronization within Pthread programs. We conducted this study using two benchmarks: the first benchmark, where each file contains one bug, and the second benchmark, where each file contains multiple bugs. We analyzed the impact of different prompt engineering techniques on detection performance and proposed a zero-shot prompt template with specific keywords to improve ChatGPT's accuracy. Additionally, we explored how dialogue-based prompting and keyword specificity further improve fault detection capabilities.

A key finding of our study is that example-based prompting significantly enhances ChatGPT's ability to recognize and understand synchronization mechanisms. Providing examples in prompts plays a crucial role in improving model performance by offering concrete scenarios for analysis. These examples help the model identify patterns associated with data race bugs, leading to higher prediction accuracy. To evaluate ChatGPT's effectiveness, we compare its performance with Cursor, another large language model that serves as a fault detection baseline.

To enhance the effectiveness of data race detection, we developed three Hybrid Prompting techniques: Hybrid Prompting 1, which integrates Chain-of-Thought (CoT) Prompting with Few-Shot Prompting; Hybrid Prompting 2, which combines Retrieval-Augmented Generation (RAG) with Few-Shot Prompting; and Hybrid Prompting 3, which applies Prompt Chaining with Few-Shot Prompting. These techniques were designed to improve ChatGPT's ability to recognize synchronization patterns and more accurately detect concurrency issues.

Our results demonstrate that Hybrid Prompting techniques consistently outperform simple prompting strategies and Cursor, the baseline model. Notably, Hybrid Prompting 3, which applies Prompt Chaining with Few-Shot Prompting, achieved the highest accuracy in the first benchmark, where each file contains one bug. This success highlights the effectiveness of Prompt Chaining, which breaks down complex tasks into smaller sub-tasks. These sub-tasks include scenarios for fixed and faulty cases, allowing ChatGPT to analyze the provided code and accurately detect data races systematically. In contrast, Hybrid Prompting 1, which integrates Chain-of-Thought (CoT) Prompting with Few-Shot Prompting, achieved the highest accuracy in the second benchmark, which consists of files with multiple bugs. This improvement is due to the advantage of CoT, which allows ChatGPT to engage in a structured reasoning

process, breaking down complex synchronization issues step by step before making a final decision. This approach is particularly beneficial in cases where multiple data races exist in the code, as it helps the model systematically analyze and identify each bug.

This study demonstrates the effectiveness of structured and example-driven prompting methods, particularly in detecting synchronization issues such as data races. We evaluated each approach based on its strengths and limitations, focusing on how ChatGPT's responses varied in identifying Pthread-related issues. Future work may extend this study to other shared-memory parallel programming models, such as OpenMP, Cilk, and Intel TBB, to assess the adaptability of prompt-based techniques. Additional directions include exploring ChatGPT's ability to detect other concurrency issues-such as deadlocks, livelocks, priority inversion, and memory leaks-and integrating LLM-based analysis with traditional static and dynamic tools. These enhancements could lead to more reliable and efficient automated fault detection systems in parallel programming.

Appendix A: Example of prompt used in the study

This appendix provides an example of Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting using a simplified example. We present the output at each step to illustrate the process. This prompt was used in our study to evaluate ChatGPT's ability to detect data race bugs in Pthread programs. It was designed with specific instructions and contextual guidance to improve detection accuracy.

```
#include <pthread.h>
#include <stdio.h>
   ......
   ......

void *thread_function(void *arg) {
   pthread_mutex_lock(&lock);
   shared_var++;
   pthread_mutex_unlock(&lock);
   return NULL;
   ......
   ......
```

Listing 7 An example of a Pthread program snippet.

```
You are an expert. Your task is to analyze the following
    parallel programming code for synchronization problems,
    such as a Data Race. If a problem exists, respond with '1';
     if none, respond with '0'.

   Check all shared variables accessed by multiple threads.

   Code: {Pthread program}

Output: [Step 1] Identifying shared variables in the provided
    code snippet...
```

Listing 8 Output Prompt Template for Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting (Prompt Template 1)

```
Are shared variables modified without proper synchronization (
    no pthread_mutex_lock/unlock)?

       Here are some examples:
            Example 1:
             Code:
                 pthread_mutex_lock(&lock);
                 Shared Variables
                 pthread_mutex_unlock(&lock);
             Response: 0

             Example 2:
             code:
                 pthread_mutex_lock(&lock);
                 Shared Variables
             Response: 1

             Example 3:
             code:
                 Shared Variables
             Response: 1

   Result from the previous step: {shared_variables}
   [Step 1] Shared Variables Identified: The shared variable in
        the provided code is `shared_var`. This variable is
       accessed by multiple threads within the `thread_function
       `. Each thread increments the value of `shared_var`.
   The code properly uses a mutex (`pthread_mutex_t lock`) to
       synchronize access to `shared_var`, ensuring that only
       one thread can modify the variable at a time. This
       prevents any data race conditions related to `shared_var
       `.

Output: `shared_var`
```

Listing 9 Output Prompt Template for Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting (Prompt Template 2)

```
1. If synchronization (pthread_mutex_lock/unlock) is used
    correctly around Shared Variables, respond with "0".
2. If no synchronization or improper synchronization is
    detected, respond with "1".

   Result from the previous step: {synchronization_check}

Output:[Step 2] Checking synchronization of shared variables...
       [Step 2] Synchronization Check Result: 0
       [Step 3] Evaluating final synchronization status...
       [Step 3] Final Evaluation Result: 0

Final Response: 0 (No Bug Detected)
```

Listing 10 Output Prompt Template for Hybrid Prompting 3: Prompt Chaining with Few-Shot Prompting (Prompt Template 3)

Acknowledgements During the paper writing process, we used Grammarly [60] for grammar checking and ChatGPT for rephrasing the title and assisting with the experiments. No funding was received to help with the preparation of this manuscript.

Author contributions This work was conceptualized, designed, and executed by MA. All experiments, data analysis, and manuscript preparation were conducted by May. LW provided supervision, guidance, and valuable feedback throughout the research and manuscript development process.

Data availability No datasets were generated or analysed during the current study.

References

1. Guo Q, Cao J, Xie X, Liu S, Li X, Chen B, Peng X (2024) Exploring the potential of chatgpt in automated code refinement: An empirical study. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3597503.3623306
2. Khojah R, Mohamad M, Leitner P, Oliveira Neto FG (2024) Beyond code generation: An observational study of chatgpt usage in software engineering practice. Proc. ACM Softw. Eng. **1**(FSE) https://doi.org/10.1145/3660788
3. Sridhara G, G, RH, Mazumdar S (2023) ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks . https://arxiv.org/abs/2305.16837
4. Nascimento N, Alencar P, Cowan D (2023) Comparing Software Developers with ChatGPT: An Empirical Investigation . https://arxiv.org/abs/2305.11837
5. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano P, Leike J, Lowe R (2022) Training language models to follow instructions with human feedback. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. Curran Associates Inc., Red Hook, NY, USA
6. Rasnayaka S, Wang G, Shariffdeen R, Iyer GN (2024) An empirical study on usage and perceptions of llms in a software engineering project. In: Proceedings of the 1st International Workshop on Large Language Models for Code. LLM4Code '24, pp. 111–118. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3643795.3648379
7. Zeng Z, Tan H, Zhang H, Li J, Zhang Y, Zhang L (2022) An extensive study on pre-trained models for program understanding and generation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022, pp. 39–51. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3533767.3534390
8. Xia C, Wei Y, Zhang L (2022) Practical program repair in the era of large pre-trained language models. ArXiv arXiv:abs/2210.14179

9. Xia CS, Wei Y, Zhang L (2023) Automated program repair in the era of large pre-trained language models. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, pp. 1482–1494. IEEE Press, ??? . https://doi.org/10.1109/ICSE48619.2023.00129
10. Fan Z, Gao X, Roychoudhury A, Tan SH (2022) Improving automatically generated code from codex via automated program repair. ArXiv arXiv:abs/2205.10583
11. Cao J, Li M, Wen M, Cheung S-C. A Study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. https://doi.org/10.48550/arXiv.2304.08191
12. Chen M, Tworek J, Jun H, Yuan Q, Pondé H, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings DW, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Babuschkin I, Balaji S, Jain S, Carr A, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight MM, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021) Evaluating large language models trained on code. ArXiv arXiv:abs/2107.03374
13. Nichols B, Buttlar D, Farrell JP (1996) Pthreads Programming. O'Reilly & Associates Inc, USA
14. Lewis B, Berg DJ (1998) Multithreaded Programming with Pthreads. Prentice-Hall Inc, USA
15. Chen L, Ding X, Emani M, Vanderbruggen T, Lin P-H, Liao C. Data Race Detection Using Large Language Models. https://doi.org/10.48550/arXiv.2308.07505
16. Sobania D, Briesch M, Hanna C, Petke J. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. https://doi.org/10.48550/arXiv.2301.08653
17. Jiang S, Zhang J, Chen W, Wang B, Zhou J, Zhang J (2024) Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose Llms, pp. 75–78 . https://doi.org/10.1145/3643795.3648390
18. Geng M, Wang S, Dong D, Wang H, Li G, Jin Z, Mao X, Liao X (2023) Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), 453–465
19. Davis E Mathematics, Word Problems, Common Sense, and Artificial Intelligence. https://doi.org/10.48550/arXiv.2301.09723
20. Gilson A, Safraneck C, Huang T, Socrates V, Chi L, Taylor R, Chartash D How Well Does ChatGPT Do When Taking the Medical Licensing Exams? The Implications of Large Language Models for Medical Education and Knowledge Assessment. https://doi.org/10.1101/2022.12.23.22283901
21. Guo B, Zhang X, Wang Z, Jiang M, Nie J, Ding Y, Yue J, Wu Y How Close Is ChatGPT to Human Experts? Comparison Corpus, Evaluation, and Detection. https://doi.org/10.48550/arXiv.2301.07597
22. Chen L, Lin P-H, Vanderbruggen T, Liao C, Emani M, Supinski B LM4HPC: Towards Effective Language Model Application in High-Performance Computing. https://doi.org/10.1007/978-3-031-40744-4_2
23. Ding X, Chen L, Emani M, Liao C, Lin P-H, Vanderbruggen T, Xie Z, Cerpa A, Du W (2023) Hpc-gpt: Integrating large language model for high-performance computing. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 951–960. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3624062.3624172
24. Bang Y, Lee N, Dai W, Su D, Wilie B, Lovenia H, Ji Z, Yu T, Chung W, Do Q, Yan X, Fung P A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. https://doi.org/10.48550/arXiv.2302.04023
25. Cahyawijaya S, Winata GI, Wilie B, Vincentio K, Li X, Kuncoro A, Ruder S, Lim ZY, Bahar S, Khodra M, Purwarianti A, Fung P (2021) IndoNLG: Benchmark and resources for evaluating Indonesian natural language generation. In: Moens, M.-F., Huang, X., Specia, L., Yih, S.W.-t. (eds.) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 8875–8898. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic . https://doi.org/10.18653/v1/2021.emnlp-main.699 . https://aclanthology.org/2021.emnlp-main.699
26. Jiang N, Liu K, Lutellier T, Tan L (2023) Impact of code language models on automated program repair. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, pp. 1430–1442. IEEE Press. https://doi.org/10.1109/ICSE48619.2023.00125
27. Hu Y, Ahmed UZ, Mechtaev S, Leong B, Roychoudhury A (2020) Re-factoring based program repair applied to programming assignments. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. ASE '19, pp. 388–398. IEEE Press. https://doi.org/10.1109/ASE.2019.00044

28. Nashid N, Sintaha M, Mesbah A (2023) Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, pp. 2450–2462. IEEE Press. https://doi.org/10.1109/ICSE48619.2023.00205
29. Zeng Z, Tan H, Zhang H, Li J, Zhang Y, Zhang L (2022) An extensive study on pre-trained models for program understanding and generation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022, pp. 39–51. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3533767.3534390
30. Xu FF, Alon U, Neubig G, Hellendoorn VJ (2022) A systematic evaluation of large language models of code. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. MAPS 2022, pp. 1–10. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3520312.3534862
31. Geng M, Wang S, Dong D, Wang H, Li G, Jin Z, Mao X, Liao X (2024) Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3597503.3608134
32. Ahmed T, Devanbu P (2023) Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3551349.3559555
33. Yang Y, Zhu Y, Chen S, Jian P (2023) Api comparison knowledge extraction via prompt-tuned language model. J Comput Lang 75:101200. https://doi.org/10.1016/j.cola.2023.101200
34. Huang Q, Yuan Z, Xing Z, Xu X, Zhu L, Lu Q (2022) Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13
35. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A., Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano P, Leike J, Lowe R Training Language Models to Follow Instructions with Human Feedback. https://doi.org/10.48550/arXiv.2203.02155
36. Christiano P, Leike J, Brown T, Martic M, Legg S, Amodei D (2017) Deep reinforcement learning from human preferences https://doi.org/10.48550/arXiv.1706.03741
37. Gozalo-Brizuela R, Garrido-Merchán E ChatGPT Is Not All You Need. A State of the Art Review of Large Generative AI Models. https://doi.org/10.48550/arXiv.2301.04655
38. Al-Bataineh OI (2024) Automated repair of multi-fault programs: Obstacles, approaches, and prospects. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. ASE '24, pp. 2215–2219. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3691620.3695287
39. Bora U, Joshi S, Muduganti G, Upadrasta R LLOR: Automated Repair of OpenMP Programs. https://doi.org/10.48550/arXiv.2411.14590
40. Joshi S, Muduganti G GPURepair: Automated Repair of GPU Kernels. https://doi.org/10.48550/arXiv.2011.08373
41. Corporation I Intel Inspector. https://www.intel.com/
42. Fu H, Wang Z, Chen X, Fan X (2018) A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. Softw Qual J 26:1–35. https://doi.org/10.1007/s11219-017-9385-3
43. Shi Z, Mathur U, Pavlogiannis A (2024) Optimistic prediction of synchronization-reversal data races. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3597503.3639099
44. Serebryany K, Potapenko A, Iskhodzhanov T, Vyukov D (2011) Dynamic race detection with llvm compiler: Compile-time instrumentation for threadsanitizer. In: International Conference on Runtime Verification, pp. 110–114 . Springer
45. Liew D, Cogumbreiro T, Lange J (2024) Sound and partially-complete static analysis of data-races in gpu programs. Proc. ACM Program. Lang. **8**(OOPSLA2) https://doi.org/10.1145/3689797
46. Choi J-D, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M (2002) Efficient and precise datarace detection for multithreaded object-oriented programs. ACM SIGPLAN Notices 37:258–269. https://doi.org/10.1145/512529.512560

47. Malakar S, Haider TB, Shahriar R (2024) Racefixer–an automated data race fixer. arXiv preprint arXiv:2401.04221
48. Shastri J, Wang X, Shivakumar BA, Verbeek F, Ravindran B (2024) Hmtrace: Hardware-assisted memory-tagging based dynamic data race detection. arXiv preprint arXiv:2404.19139
49. Kadosh T, Schneider N, Hasabnis N, Mattson T, Pinter Y, Oren G Advising OpenMP Parallelization Via a Graph-Based Approach with Transformers. https://doi.org/10.48550/arXiv.2305.11999
50. Chen L, Bhattacharjee A, Ahmed N, Hasabnis N, Oren G, Vo V, Jannesari A OMPGPT: A Generative Pre-trained Transformer Model for OpenMP. https://doi.org/10.1007/978-3-031-69577-3_9
51. Kadosh T, Hasabnis N, Soundararajan P, Vo V, Capota M, Ahmed N, Pinter Y, Oren G OMPar: Automatic Parallelization with AI-Driven Source-to-Source Compilation. https://doi.org/10.48550/arXiv.2409.14771
52. Mišć M, Dodović M (2024) An assessment of large language models for openmp-based code parallelization: a user perspective. J Big Data 11:1019. https://doi.org/10.1186/s40537-024-01019-z
53. Alsofyani M, Wang L (2024) Detecting data races in openmp with deep learning and large language models. In: Workshop Proceedings of the 53rd International Conference on Parallel Processing. ICPP Workshops '24, pp. 96–103. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3677333.3678160
54. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D (2022) Chain-of-thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903
55. Gao Y, Xiong Y, Gao X, Jia K, Pan J, Bi Y, Dai Y, Sun J, Guo Q, Wang M, Wang H (2023) Retrieval-augmented generation for large language models: A survey. ArXiv arXiv:abs/2312.10997
56. Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Kuttler H, Lewis M, Yih W-t, Rocktäschel T, Riedel S, Kiela D (2020) Retrieval-augmented generation for knowledge-intensive nlp tasks. ArXiv arXiv:abs/2005.11401
57. POSIX threads: POSIX Threads Programming. Accessed: 2023-11-06 (n.d.). http://www.csc.villanova.edu/~mdamian/threads/posixthreadslong.html
58. Carnegie Mellon University: POSIX Threads Overview. Accessed: 2023-11-29 (n.d.). https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
59. Cursor: Cursor: AI-Powered Code Editor (2024). https://www.cursor.com/
60. Grammarly: Grammarly: Free AI Writing Assistance. Accessed: 2024-11-19 (2024). https://www.grammarly.com/