AAS

**ABSTRACT:**12313

**Keywords:** 45646

#####主體架構參考paper#####

# 參考文獻

我要寫的是，不同語言模型，然後給不同prompt，去比較生成結果

# Introduction(新版)

近年來，自動程式碼生成在LLM與自然語言處理(NLP) 領域中日益受到重視，GPT-4、Cloude、Genmini等模型能在自然語言提示中生成程式碼任務已被證實具有成功性(Chen et al., 2024)。LLM能夠從直觀的自然語言輸入中產生程式碼，為開發者提供了有前景的應用途徑，有望簡化程式設計流程並加速軟體開發週期(Haque, 2025)。除了程式碼，LLM還能夠產生自然語言文本，並提供即時且具情境關聯的回應，可應用於寫作輔助、補充教材、程式語言學習指引(Steiss et al., 2024)。

LLM之所以備受關注，主要來自其在多種應用情境中的高適應性與即時回應能力。第一，LLM能夠提供隨需支援，讓使用者在程式開發過程中隨時獲得協助，特別適用於自我導向學習或獨立開發的場景（Cao & Dede, 2023）。第二，LLM具備因應不同學習風格的潛力，能透過多種方式（如自然語言說明、簡明摘要、逐步教學）呈現內容，進一步提升使用者的學習效率與實作表現（Pesovski et al., 2024）。第三，LLM也展現出在程式碼撰寫上的輔助價值，能協助進行程式碼補全、除錯建議與重構優化，為程式教育與實務開發帶來新的可能性（Khazanchi & Khazanchi, 2024）。

相較於過往的程式理解與修復技術，LLM在程式碼生成方面展現出顯著進步。Rasnayaka et al.（2024）、Xia et al.（2022）與 Zeng et al.（2022）等學者指出，LLM具備識別與修正原始碼錯誤的能力，並能理解程式語意進行有效生成，整體效能優於傳統方法。Xia et al.（2022）更強調，預訓練語言模型在程式修復任務中，能精準執行完整函式生成、程式碼填空與單行修正，並展現出跨程式語言與多樣化開發環境下的強大泛化能力。Cao et al.（2025）進一步探討不同提示設計對除錯成效的影響，透過引入更具複雜性與依賴性的錯誤程式，提供對LLM在真實除錯任務中的完整評估。然而，儘管LLM在程式碼生成上已有實質突破，其表現仍難以與自然語言生成任務相提並論（Espejel et al., 2023）。生成可執行且語意正確的程式碼相較於文字更具挑戰性，主要原因在於語言模型必須精確對應自然語言指令，同時遵守嚴格的語法與語意規則（Wong et al., 2023）。即便是微小的錯誤，例如漏寫分號、括號或冒號，皆可能導致程式無法編譯或產生邏輯錯誤，顯示出程式碼生成對模型的語言理解與邏輯推理能力提出更高要求。

此外，

目前的 LLM 雖然表現優異，但許多為封閉源碼，或因需仰賴強大的圖形處理器（GPU）進行推論而面臨重大限制。這種依賴產生兩大挑戰：可及性與成本，特別影響個人和小型機構。高效能 GPU 價格昂貴，並不普遍存在於一般電腦設備中，限制了誰能有效運行這些模型。取得與維護這些專用硬體的經費負擔沉重，可能導致資源被分流，影響其他關鍵活動的進行，使得許多人無法充分參與最先進的 AI 研究與開發，最終限制了這些強大工具的廣泛應用與創新。

此外，即使部分 LLM 可免費使用，仍存在隱私洩露的風險(Wu et al., 2023) 為解決這些問題，研究社群開始轉向參數較少的模型。例如，儘管PALM(Chowdhery et al., 2023)、GPT-3(Brown et al., 2020)、與GPT-4(OpenAI, 2023)等大型模型分別擁有 1750 億、5400 億與 1.7 兆個參數。小型模型如 Mistral(Jiang et al., 2024) 與 LLaMa(Dubey et al., 2024) 則僅以 70 億參數展現出相近甚至更佳的效能。

此外，量化技術的興起也有助於加速模型推論。例如Activation-aware Weight Quantization(AWQ)( Lin et al., 2024)與Generative Pre-trained Transformers Quantization(GPTQ)( Frantar et al., 2022) 等熱門技術，能有效壓縮 LLM，同時保留在下游任務中的準確度。為了使用這些技術，像LlamaCPP(Khosravi, 2024) 這樣的函式庫支援在 CPU 上運行已量化的 LLM。LlamaCPP 專注於 GGUF（GPT-Generated Unified Format）檔案，這是一種優化儲存與推論效率的進階二進位格式。

儘管優點顯著，LLM在教育應用中仍面臨若干挑戰與限制。最主要的疑慮在於生成內容的品質與正確性。由於LLM是在龐大資料集上訓練而成，資料集本身同時涵蓋正確資訊與錯誤資訊，因此有時會生成不準確或具偏見的回應(Doughty et al., 2024)。如果模型缺乏深層語境理解，往往會往產生空泛或無關的資訊(Weisz et al., 2024)。為了處理上述挑戰，一種解法是使用優質資料與人為回饋來微調LLM，以提高其準確性與情境對應能力(Dave et al., 2023)。另一種方法是透過提示工程（Prompt Engineering），透過Prompt設計，給予精確且具情境性的輸入引導模型回應(Alier et al., 2024)。零樣本（zero-shot）與少樣本（few-shot）提示法可以讓模型在極少的訓練資料下執行新任務，善用既有知識產出準確答案(Brown et al., 2020)。Chain-of-Thought（CoT）提示法透過將任務拆解為邏輯步驟來提升模型推理能力，在處理複雜問題時能增進準確性(Zhang et al., 2022)。

(結尾放)

本研究的動機是展現對CPU 友好模型在從自然語言生成Pthread程式碼任務中的潛力與可靠性，因此我們評估了這些開源模型的表現，並將其與封閉源模型進行比較，以檢視其能力與限制。為求完整性，此外，我們還提出一個 Chain-of-Thought（CoT）提示詞以提升這些模型的效能。提示詞會指定模型的角色、需要關注的關鍵詞，並描述目標任務。接著提供一個示例進行驗證（一次性推論），最後明確指出目標是產出唯一的解題函數，不需額外解釋或註解。

（研究問題）

本研究以作業系統課程中的 Pthread 教學為場域，探討 ChatGPT 所生成之 Pthread 程式碼在教學上的應用潛力與侷限。我們提出兩個核心研究問題：

RQ1：在作業系統課程中，LLM 所生成之 Pthread 程式碼與人為撰寫之程式碼在正確性、語意清晰度與系統效能上有何差異？

RQ2：若導入 Chain-of-Thought（CoT）提示優化策略，是否能有效改善 LLM 在平行程式生成中的執行效率、記憶體使用量與峰值記憶體表現？

我們特別聚焦於偵測並修復平行程式中的資料競爭錯誤，原因有二：第一，平行程式語言與序列式語言在結構上有顯著不同，其設計目的在於同時於多個處理單元中執行多個任務，並內建資料共享與同步機制。在此類程式中，資料競爭錯誤會導致不可預測的行為與錯誤，且極難重現與除錯，因為此類錯誤發生於多個執行緒未正確同步地存取共享資料時，可能導致資料損毀或系統崩潰(Nichols, 1996)。第二Pthread 提供低階控制的執行緒管理與同步功能，特別適合用於研究像資料競爭與同步錯誤等複雜的並行問題。據我們所知，目前尚無研究探討 LLM 在分析 Pthread 程式方面的應用。藉由聚焦於 Pthread，我們期望填補此研究空白，並為 AI 在平行程式設計中的應用提供新的見解。

本研究圍繞三個核心研究問題展開：

RQ3. ChatGPT 能有效除錯 Pthread 程式嗎？　本題旨在透過基本提示進行錯誤偵測，評估 ChatGPT 對 Pthread 程式的除錯表現，並以準確率、F1 分數、召回率與精確率進行量化評估。

RQ4. 對話互動是否能協助錯誤偵測？由於 ChatGPT 是設計為互動式工具，其是否能透過使用者對話互動提升 Pthread 程式錯誤偵測的效能仍屬未知。因此，我們探討在提示中加入錯誤類型（如資料競爭）與執行緒同步資訊，是否能進一步提升其偵錯效果。

RQ5. 提示工程如何改善 ChatGPT 的錯誤偵測效能？研究已指出提示工程能顯著提升 LLM 的回答品質，但如何設計最佳提示仍是一大挑戰。我們將探討多種提示策略，包括 Zero-Shot Learning、Few-Shot Learning、Chain-of-Thought 與 Retrieval-Augmented Generation，並發展出數種混合提示模板以歸納成效最佳之提示設計，並應用於基準資料集驗證其效果。

# Related Work01：LLM for software engineering

大型語言模型（LLMs）已在多個領域展現出變革性的潛力，包括軟體工程、數學(Guo, et al., 2023)、高效能運算(HPC)( Chen, et al., 2023) 以及自然語言處理(Cahyawijaya, et al., 2021)。在軟體工程領域中，LLMs 被廣泛應用於程式修復、程式碼摘要與程式碼生成(Ahmed & Devanbu, 2022)。

以高效能運算(HPC)來說，LLM在HPC 研究中也受到越來越多關注。例如Chen, et al., (2023) 開發了模型LM4HPC，應用於程式碼生成與效能優化。Ding et al., (2023)開發出了基於 LLaMA模型的HPC-GPT，透過對 HPC 領域的問答資料進行微調，以擴展 LLM 在高效能運算領域的應用能力。

同時，LLMs 也被用於解決 API 相關的挑戰，例如偵測 API 誤用與回答 API 相關問題。例如Yang et al., (2023) 開發了基於LLM得模型APICKnow，用於從文本中擷取 API 實體及其語意關係。該研究使用 Stack Overflow 資料集進行實驗，證實 APICKnow 無需大量標註資料即可達到高準確率。Huang et al., (2022)則提出透過微調 code-masked 語言模型（MLM），來解決程式碼片段中 fully qualified name（FQN）解析的問題。這些研究不僅凸顯了 LLM 的能力，也指出其可改進之處，並提供實際應用上的洞見。

Cao et al., (2025) 評估了 ChatGPT 在深度學習程式修復上的能力，透過不同提示設計檢測與修復錯誤。亦強調預訓練語言模型在程式修復方面的優勢，尤其是在跨程式語言與程式碼模式上的泛化能力。

在與程式任務相關的少樣本學習程序中，Nashid et al., (2023) 提出了一種基於檢索的提示選擇方法，引導語言模型進行程式碼修復與測試斷言生成。該方法顯著提升了表現，凸顯了檢索式提示策略在強化少樣本學習中的價值。Ahmed & Devanbu (2023) 聚焦於專案特定的程式碼摘要生成，利用具有專案脈絡的範例進行少樣本訓練，以微調 LLM，進而提升摘要的相關性與品質。

# Related Work01：LLM應用於平行程式

透過多核心處理器的應用，平行程式設計（Pthreads）允許多個執行緒同時運行，進而提升程式效能。儘管執行緒管理能帶來諸多效益，但同時也可能引發如並行錯誤等挑戰，尤其是資料競爭（data races）。當多個執行緒在未正確同步的情況下同時存取同一變數時，就可能發生資料競爭，進而導致不可預期的行為甚至系統當機。為避免此類問題，程式中會實作同步機制，例如條件變數（condition variables）與互斥鎖（mutex locks），以確保執行緒安全。

要解決資料競爭問題，需要有效的偵測與修復方法。過去與近期的研究提出了多種偵測與修復策略，包括靜態分析（static analysis）、動態分析（dynamic analysis）與執行階段驗證（runtime verification）等方法。(Shi et al., 2024)。這些技術可用來偵測，甚至在某些情況下修復資料競爭。Serebryany et al.,(2011) 提出一種整合於 LLVM 編譯器的動態競爭偵測方法，透過編譯時期插裝技術強化 ThreadSanitizer 的功能。Liew et al., (2024) 則針對 GPU 程式提出一種健全但部分完備的靜態分析技術，結合靜態分析與領域特定的抽象來應對 GPU 並行性的挑戰。Choi et al., (2022) 結合靜態與動態分析，發展出一種能精確偵測多執行緒物件導向程式中資料競爭的方法。Malakar et al., (2024)提出自動化工具RaceFixer，能分析並行問題並推薦適當的同步機制（如互斥鎖）來修復資料競爭。例如

雖然已有多種技術用於偵測與修復資料競爭，但 LLM具備獨特優勢，使其在現代平行程式設計中極具價值。近年來，研究顯著推進了人們對 LLM 在平行程式設計中應用潛力的理解。Kadosh et al., (2023) 使用源碼的圖形表示結合 Transformer 模型，預測 OpenMP 的 pragma 並分析迴圈的平行化潛力。OpenMP（Open Multi-Processing） 是一個 API，可讓開發者在共享記憶體多平台上撰寫平行程式碼。Chen et al., (2024)提出OMPGPT一款專為此領域設計的 LLM，能協助產生 OpenMP 的 pragma，展現出領域特定 LLM 在自動平行化方面的潛力。包括 ChatGPT 與 GitHub Copilot 在內的工具，也已被應用於平行程式碼生成(Mišić & Dodović, 2024)及OpenMP 並行錯誤偵測的研究(Alsofyani & Wang, 2024)。這類研究凸顯 LLM 在平行程式設計中的重要性，特別是在提供智慧程式建議與自動化複雜任務（如迴圈平行化與語法結構生成）方面的優勢。LLMs 能適應多種程式設計範式，並從龐大的程式碼資料庫中持續學習，使其能夠不斷演化，並有效因應快速變化的技術環境。結合其高可用性與能整合錯誤偵測、修復與解說等任務的能力，使得 LLMs 成為推進平行程式設計與培育未來開發者的關鍵工具。

# Related Work01：評估程式碼

程式碼效率常見的衡量指標包括記憶體使用量、CPU使用率、執行時間（Run Time）與程式碼複雜度。Dou et al., (2024)與Hou and Ji (2025)比較了LLM與人類程式設計師的表現。發現GPT-4 雖然在結構化任務中表現優異，但在面對模糊任務時表現不佳，而人類設計師撰寫的程式碼在執行時間與記憶體使用方面通常更有效率，突顯出 LLM 與人類解法在優化方面的落差。Niu et al., (2024)的研究中，除了正確性外，研究也納入執行時間作為效率指標，並在 HumanEval、MBPP 這些基準資料集中(聚焦 Python 問題)，與LeetCodeEval資料集（涵蓋 C++ 題目）上進行測試。結果指出，LLM的執行效率仍落後於人類所撰寫的程式碼。

然而，這些先進神經網路工具的預測性質也引發了跨領域的可信度疑慮，例如自然語言處理(Deanda et al., 2015)、電腦視覺(Zhang et al., 2019)、自動駕駛(He et al., 2024)與醫療保健領域(Deanda et al., 2025)。程式碼生成的脈絡中，主要關注於LLM所生成程式碼的實際可用性，尤其在硬體受限下的環境更需重視效能指標，例如記憶體使用量、CPU使用率、執行時間與資源消耗(Coignion et al., 2024; Qiu et al., 2024)。

為了評估生成程式碼的正確性，即是否達成預期功能。Paul et al.,(2024)分析了現有的基準資料集。例如HUMANEVAL(Chen et al., 2021)、MBPP(Austin et al., 2021)等方式來評估LLM所產出的程式碼。相關研究採用pass@k作為效能評估指標。pass@k意旨在k次生成中至少成功一次的機率。然而Paul et al.,(2024)所研究的重點為 LLM 所產生程式碼的正確性，但對效率問題則未多加關注。Du et al., (2024)則提到，程式碼的效率應該被定義在資源消耗最小的情況下完成任務的能力。這樣的程式碼才能提升使用者體驗、節省能源，並使應用更具成本效益。

儘管近期的基準研究開始重視程式碼的功能正確性，像是EFFIBENCH是一套專門用來評估LLM生成程式碼正確性與效率的測試集(Huang et al., 2024)。但研究也指出，即使LLM生成的程式碼在功能上是正確的，卻經常在記憶體使用與執行時間方面不如人類撰寫的程式碼(Huang et al., 2024)。此結果源自目前主流的訓練範式，即「下一個程式碼 token 預測」傾向於強調局部模式的補全，而非尋找整體上更有效率的演算法結構(Guo et al., 2024)。儘管語法上正確，但局部補全的方法會產生多餘的程式碼區塊，導致時間與空間複雜度上升，進而造成計算資源的浪費。LLM生成程式碼的固有低效率，會使應用程式的執行速度變慢、運作成本上升，甚至無法符合系統需求。尤其是IoT、邊緣運算系統、雲端裝置等資源受限的環境下部署時，記憶體使用量與執行速度為關鍵考量(Bolón-Canedo et al., 2024; Solovyeva et al., 2025)。

在訓練資料集上，同時也存在有效與無效的程式碼，這會讓模型對於提示語意產生幻覺(hallucination)，進一步導致不必要的能源消耗與在關鍵應用場景下的效能下降(Huang et al., 2025)。Ehsani et al., (2025)與Khojah et al., (2025)也指出，若prompt本身含糊或結構不良，模型可能誤解任務本意，進而產出不必要複雜或效率低落的解決方案。

# Related Work01：Open-source model

開源模型也作為私有模型的替代方案逐漸浮現。這些開源模型在程式碼生成任務中同樣展現出卓越的表現。例如，InCoder(Fried et al., 2022) 與 StarCoder(Li et al., 2023) 是專為程式碼訓練的模型。有別於其他編碼模型。，InCoder 在訓練階段採用了「因果遮蔽」（causal masking）目標此方法結合了因果語言模型與遮蔽語言模型的優點，增強了其學習能力(Devlin et al., 2019)。相對地，StarCoder 在訓練中引入了 FlashAttention 機制，此技術能加速注意力運算並降低記憶體使用，進而優化模型效能。

Roziere et al(2023)發布了Code Llama，使用因果遮蔽技術來訓練補全模型。該模型基於 LLaMA-2進行微調，並引入了一個「長上下文微調」（Long Context Fine-Tuning，LCFT）階段使模型能處理長達 16,384 個詞元的序列，相較於早期 LLaMA-2 階段所支援的 4,096 詞元是一大進步。儘管 Code Llama僅有 70 億參數，但其在 HumanEval與MBPP資料集上的表現已超越了參數高達 700 億的 LLaMA 模型，顯示其在處理大規模程式碼挑戰方面的效率與效能(Yu et al., 2024)。然而，LLaMA-2的效能已被更新的 LLaMA-3(Curto et al., 2024)與LLaMA-3.1(Dubey et al., 2024)。值得注意的是，LLaMA-3.1 的效能甚至優於 LLaMA-2，這歸因於多項改進：它是一個多語言模型，在預訓練與後訓練階段都受益於更高品質與更大量的數據，且其模型參數規模也更大。

除了持續朝向開源語言模型的努力外，Phi 模型的推出也顯示了在模型規模上競爭的興趣日益增加(Gunasekar et al., 2023)。最初的模型 Phi-1是一個基於 Transformer 的模型，擁有 13 億參數，專為程式碼任務設計，並強調使用教科書品質的訓練資料(Gunasekar et al., 2023)。改進版本 Phi-1.5(Li et al., 2023) 在 Phi-1 的基礎上加入了下一詞預測目標，並使用數十億詞元的資料集進行訓練。Phi-2是一個擁有 27 億參數的模型，使用 Phi-1.5 的訓練資料結合新的合成 NLP 文本與篩選過的網站資料進行訓練(Javaheripi et al., 2023)。

# Related Work01：Small language models

受到 LLaMA-1 以及 ChatGPT2、Claude3 與 Microsoft BingChat4 等指令遵循模型成功的啟發，Taori et al., (2023) 推出了 Alpaca。Alpaca 是從 LLaMA-1-7B 微調而來的 70 億參數模型，使用了 52,000 筆指令遵循示例。儘管模型規模遠小於對手，Alpaca 的行為表現卻與 OpenAI 的 text-davinci-003 相似。

值得一提的是，Chiang et al., (2023)推出的Vicuna結合了 LLaMA 與 Alpaca 的優勢，其參數數量略高於 Alpaca，達到 130 億。雖然上述模型的參數量最高為 130 億。Jiang et al., (2024)推出了突破性的模型 Mistral。Mistral 是一個擁有 70 億參數的語言模型，其使用 Grouped-Query Attention（GQA）提升推論速度(Ainslie et al., 2023)。並透過 Sliding Window Attention（SWA）有效處理任意長度的序列，降低成本(Beltagy et al., 2020)。Mistral 的表現超越了 LLaMA-2 13B在所有測試基準上的成績，包括程式碼生成任務，甚至優於已釋出的 LLaMA-2 34B 變體(Touvron et al., 2023)。基於 Mistral 的成功，改進版 Mixtral 8x7B採用了稀疏專家混合（Sparse Mixture of Experts，SMoE）架構。這項改進使 Mixtral 在所有基準上超越或匹敵 LLaMA-2 70B 與 GPT-3.5 模型的效能，同時仍採用與 Mistral 7B 相同的基本架構。其主要差異在於結構設計：Mixtral 每層包含八個前饋區塊（稱為「專家」），提升了模型的處理效率與回應能力(Jiang et al., 2024)。

Zephyr是 Mistral-7B 的升級版本(Tunstall et al., 2023)，透過 ∼20 萬筆 UltraChat 資料集(Ding et al., 2023) 進行微調，並進一步利用約 6.4 萬筆 UltraFeedback 資料集進行對齊(Cui et al., 2023)。該模型在 MT-Bench 基準上超越了 LLaMA2 70B。Hu et al., (2024)提出的MiniCPM 模型在 MT-Bench 表現上顯著超越了 Zephyr-7B-alpha。與 Zephyr-7B 類似，MiniCPM 同樣使用監督式微調與 DPO 演算法，但其模型更為精簡，僅有 20 億參數。

# Related Work02：提示工程（Prompt Engineering）

提示工程（Prompt Engineering）的發展歷程始於「零樣本提示（Zero-Shot Prompting）」，這種方法讓模型在沒有特定訓練資料的情況下，僅依據一般性的指令執行任務。隨後「少樣本提示（Few-Shot Prompting）」的出現引入少量範例，以協助模型更準確地理解任務內容(Brown et al., 2020)。Chain-of-Thought（CoT）由Wei et al., (2022) 提出，進一步推進提示技術，透過逐步推理的方式，顯著提升模型在推理任務中的表現。Niu et al., (2024) 採用了不同的提示設計技巧，例如針對簡單問題使用簡易提示（simple prompt），針對複雜問題則使用Chain-of-Thought（CoT）提示。CoT將問題分解為邏輯步驟，藉此減少冗餘、提升效率。結果顯示，CoT 可使執行時間降低 15%，記憶體使用量降低 12%，是一種實用的優化工具(Wei et al., 2022)。

在自動化流程上，Zhang et al., (2022) 提出了「自動化 Chain-of-Thought（Auto-CoT）提示」，能自動產生多樣化的推理鏈，在無需人工設計的情況下提升準確度與穩定性。Wang et al., (2022) 在 CoT 提示的基礎上提出「自我一致性（Self-Consistency）」機制，透過產生多條推理路徑，並選出最一致的答案，藉此進一步提升解題準確性與穩健性。Zhao et al., (2023) 提出了「邏輯推理提示（LogiCoT）」，將符號邏輯原則引入推理過程，用於驗證每個推理步驟的正確性，以降低錯誤與幻覺（hallucination）的發生。

而為了有效實作上述提示技術，研究者也設計了多種提示框架，使這些方法得以實際應用於現實世界中。提示框架在銜接模型能力與使用者實際需求之間，扮演了關鍵橋樑的角色。這些框架提供必要的基礎設施，如整合外部工具、維持歷史資訊、確保輸出結果具備結構性與安全性(Liu et al., 2023)。例如LangChain 與 Semantic Kernel 等框架可讓 LLM 與資料庫、網頁瀏覽器及其他外部系統互動，進而克服 LLM 的內在限制並擴展其應用範疇(Liu et al., 2023)。透過一個全面而可調適的結構，全面提升人機溝通的品質（Lo, 2023）。

# Study design

我們開發了一套實驗流程，採用 OpenAI 提供的 ChatGPT API，明確使用 **GPT-4-turbo 模型**。為了處理 ChatGPT 回應中具有非決定性（non-deterministic）的特性，我們對每一個查詢進行了三次獨立執行，每次皆重新啟動一個新的 API 對話會話。整個過程中，我們完整記錄了所有提示語（prompts）及其對應的回應內容。

我們套用軟體工程研究中常用的評估指標，計算每個提示的 準確率（accuracy）、F1 分數（F1 score）、精確率（precision）與召回率（recall）。最後，我們對三次試驗的指標取平均值，以提供一個更全面的錯誤檢測效能評估。

## 資料來源與背景說明

在平行程式設計中，Pthreads（POSIX Threads）允許開發者透過多個執行緒在多核心處理器上同時執行任務，以提升效能與資源利用率。但並行執行所帶來的共享資源問題亦隨之增加，尤其是資料競爭（data race）的風險，是平行程式教學與開發中極具挑戰性的議題。data race發生於兩個或以上的Thread。在缺乏適當同步機制的情況下同時讀寫共享變數，導致資料不一致、毀損，甚至系統不穩定。為了避免這樣的錯誤，互斥鎖（mutex）成為同步控制的主要工具，核心在於確保同一時刻僅有一個執行緒可操作共享資源。其中pthread\_mutex\_lock() 用於鎖定 mutex 物件，使得該執行緒獲得資源排他權；而pthread\_mutex\_unlock() 則釋放鎖定，允許其他執行緒進行後續操作。此種明確的同步模式，有助於建立可預測與穩定的平行程式邏輯。

本研究的基準資料集來自GitHub 上所開源的平行程式碼檔案[[1]](#footnote-1)。這些程式皆包含明確的 mutex 同步操作，尤其聚焦於 pthread\_mutex\_lock / pthread\_mutex\_unlock 的使用樣式與順序。我們從原始程式碼中擷取出具教學與測試價值的範例，並依照同步結構與執行緒模型將其分類為兩個子集合。每個程式範例長度介於 50 至 250 行之間，涵蓋常見的資料共享模式、同步邏輯與錯誤案例。此資料集不僅提供 LLM 程式生成能力評估之依據，更具備高度的教學應用潛力，可作為教材、練習題或錯誤診斷練習的基礎。

## 基準集建構與錯誤資料設計

為深入評估研究上LLM在辨識與理解平行程式錯誤(特別是data race) 方面的能力，本研究以兩組測試集作為基準，分別為單一資料競爭錯誤（Single-Error Dataset）與多重資料競爭錯誤（Multi-Error Dataset）：

* 單一資料競爭錯誤（Single-Error Dataset）包含 22 對（共 44 筆） Pthread 程式碼樣本，每一對包含一個正確實作同步機制的版本（fixed）與一個省略關鍵同步語句的錯誤版本（faulty）。錯誤版本僅包含一種類型的資料競爭錯誤，確保錯誤來源明確，便於觀察模型在單一錯誤辨識情境下的回應能力與診斷正確率。進一步地，為探討資料分佈對 LLM 輸出品質的影響，我們採用以下兩種資料分佈方式進行模型測試：
  + 平衡資料集（Balanced Dataset）：錯誤程式與修正程式數量相等
  + 不平衡資料集（Imbalanced Dataset）：錯誤程式與修正程式數量比例失衡，用以模擬實際教學或開發場景中錯誤樣本稀少或過多的情境。
* 多重資料競爭錯誤（Multi-Error Dataset）：此子集亦包含 22 對（共 44 筆） 程式樣本，但錯誤版本設計為包含兩至四種 mutex 同步錯誤，例如：多處遺漏鎖定、錯誤釋放順序、或跨區塊競爭。這類設計更接近學生常見的初學者程式錯誤，也更能考驗 LLM 在處理非單一因果鏈的錯誤推理能力。

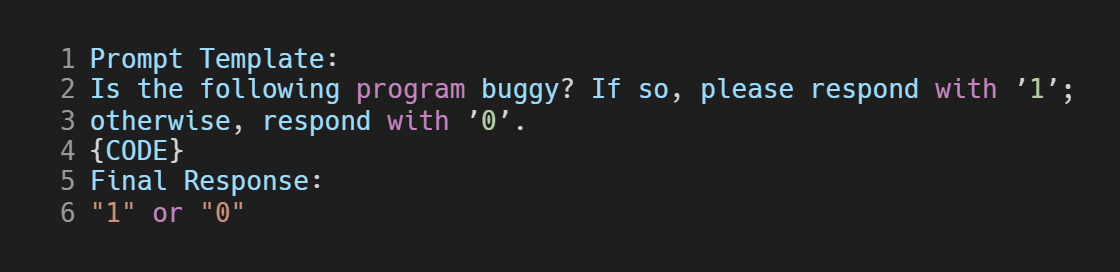
透過這樣的基準設計，我們得以系統性地評估不同提示技術在偵測單一與多重資料競爭錯誤時的效能。此架構可全面檢視 ChatGPT 在處理並行錯誤（如資料競爭）方面的能力。本研究所使用的程式樣本皆以 Pthreads 為基礎實作，並以省略特定同步機制（如 pthread\_mutex\_lock() 與 pthread\_mutex\_unlock()）的方式，人為設計具資料競爭風險的錯誤版本。對應地，每個錯誤版本皆有一個經手動修正、邏輯正確的對應程式，構成錯誤–修正配對樣本（faulty-fixed pair），以利比較分析。

# Result

## RQ1:以baseline 與 Zero-shot作為子變項，分析LLM 在錯誤識別準確率與說明品質上是否存在顯著差異？

為了評估LLM在我們所設計的基準測試下，對平行程式進行除錯的能力。本研究給出的prompt如圖XX所表示。本模板包含一段 CODE 區塊，我們將每個已修復與有錯的程式碼範例依序插入該區塊中。

*Table 1*: Prompt Template for Basic Prompt (Baseline)



結果指出，LLM系統在三次執行中的程式碼都標記為錯誤，並在基準集中所有檔案都回傳值「1」。取得結果後，我們進一步要求LLM系統說明程式碼標記錯誤的理由。然而，在多次情況下，LLM系統所給的解釋不夠令人滿意。由於 Pthread 程式的複雜性，LLM系統常給出較為籠統或模糊的描述，這顯示其在缺乏足夠上下文的情況下，無法精準地識別特定錯誤。以下是LLM系統回應的部分示例：

**Response**: “*There are multiple possible issues in the given program that may cause runtime errors.*”

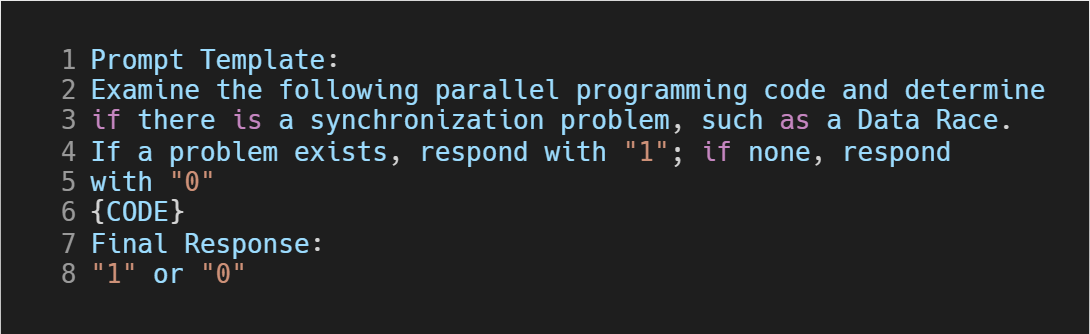
**Response**: “*The way the mutex is used in the program may result in a synchronization issue.*”

**Response**: “*The error in the program stems from its dynamic initialization process.*”

**Response**: “*The fault in the program is caused by improper thread termination.*”

接著，本研究嘗試設計Zero-Shot prompt，即在prompt中加入“parallel programming codes”、 “data race”、 “synchronization problem”等關鍵字，並指示LLM系統對所有程式碼進行檢查。這個方法的核心，是讓 ChatGPT 在**沒有任何範例的前提下**，執行將錯誤程式與正確程式區分的任務。如如圖XX所表示。

Figure : Prompt Template for Zero-Shot Prompt Template



結果指出，我們測試了多種關鍵字組合對LLM系統的影響，並觀察到準確率的提升。當使用像是” parallel programming”、” synchronization problem”等精確關鍵詞時，LLM系統的表現會更好，相較於RQ1，若prompt能明確指出要辨識同步問題LLM系統較能提供正確的回應，並能更準確地偵測錯誤。此外，當在第一組與第二組基準測試中使用 Zero-Shot Learning Prompt 時，ChatGPT 的錯誤偵測準確率約提升了 61%。

## RQ2 :提示工程（Prompt Engineering）如何協助提升 ChatGPT 的錯誤偵測效能？

在RQ1中，雖然透過Zero-Shot prompt的方式可以提升精確度，但仍還有待改進空間。因此我們採用了多種進階提示技巧，包括：

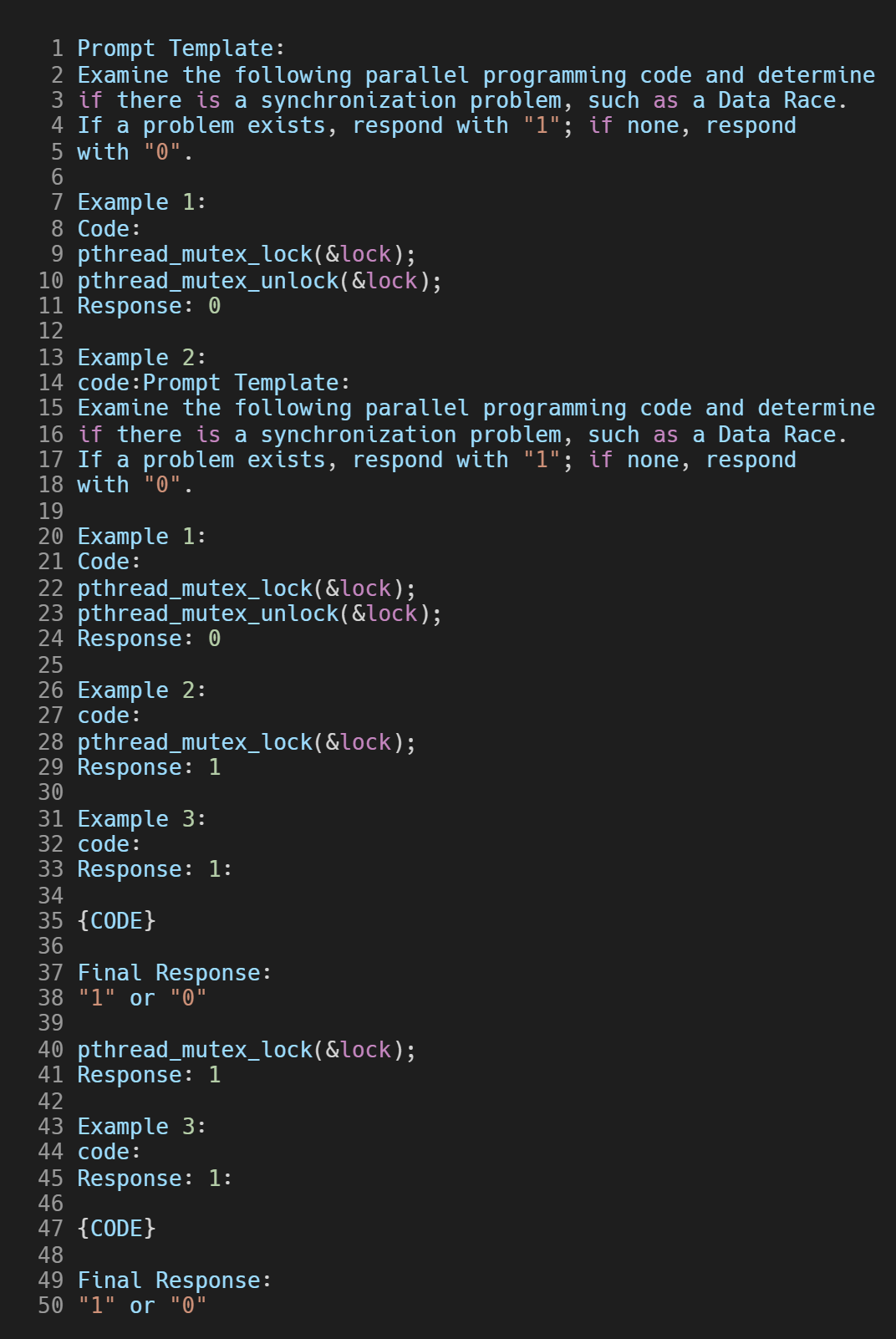
* Few-Shot Learning Prompt（少樣本學習提示）
* Chain-of-Thought Prompt（思路鏈提示）
* Retrieval-Augmented Generation (RAG) Prompt（檢索增強生成提示）

我們基於這三種提示技巧提出三種混和提示策略：

* Hybrid Prompting 1：結合 Chain-of-Thought 與 Few-Shot Prompting
* Hybrid Prompting 2：結合 Retrieval-Augmented Generation 與 Few-Shot Prompting
* Hybrid Prompting 3：結合 Prompt Chaining 與 Few-Shot Prompting

在

Figure : Prompt Template for Few-Shot Learning Prompt



# Conclusion

對於LLM是否能有效除錯程式碼(RQ1)，Pthread程式本身涉及複雜的並行性問題，例如data race、synchronization 與deadlock。這些都需要對Thread互動有深入理解。儘管LLM在預訓練上有接受過一般除錯概念的訓練，但其知識可能未涵蓋如 Pthread 這類特殊領域，特別是在同步原語（synchronization primitives）方面。當提示中未明確指出與平行程式相關的細節時，我們開發出的LLM系統傾向提供通用型敘述，因缺乏上下文而難以準確識別具體錯誤。若期望提升LLM在Thread分析方面的效能，可採用更具針對性的提示設計。

1. The benchmark can be found at <https://github.com/MayAlsofyani/Pthread-Benchmark> [↑](#footnote-ref-1)