TIPS Framework

**ABSTRACT:**12313

**Keywords:** 45646

#####主體架構參考paper#####

**TIPS = Template-Integrated Prompting System**

我這篇

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，隨著大型語言模型（Large Language Models, LLM）與自然語言處理（Natural Language Processing, NLP）技術的迅速發展，自動程式碼生成已成為研究與應用領域中的關注焦點。GPT-4、Claude、Gemini 等模型已被證實能透過自然語言提示成功執行程式碼生成任務（Chen et al., 2024），為開發流程帶來重大轉變。LLM 具備從直觀語言輸入產生結構化程式碼的能力，不僅簡化了程式設計流程，更有潛力顯著加速軟體開發週期（Haque, 2025）。此外，這類模型亦能生成自然語言文本，提供具情境關聯的即時回應，廣泛應用於寫作輔助、教材補充與程式語言學習指引等情境（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)。

相較於傳統的程式理解與修復技術，LLM 在程式碼生成方面展現出顯著優勢。多項研究指出，LLM 不僅能識別與修正原始碼錯誤，亦能掌握程式語意以產生有效程式碼，整體效能優於以往方法（Rasnayaka et al., 2024；Xia et al., 2022；Zeng et al., 2022）。Xia et al.（2022）進一步強調，預訓練語言模型於程式修復任務中表現出色，能精確執行完整函式生成、程式碼補全與單行修正，並展現跨語言與多開發環境的高度泛化能力。Cao et al.（2025）則探討提示設計對除錯效能的影響，藉由引入高依賴性與複雜性錯誤程式，有效評估 LLM 於真實除錯任務中的應用表現。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本生成更具挑戰性，主因在於模型需同時滿足自然語言指令的準確性與程式語法、語意的嚴格要求（Wong et al., 2023）。即便是細微錯誤，如缺漏分號、括號或冒號，皆可能導致編譯失敗或邏輯錯誤，顯示出程式碼生成對語言理解與邏輯推理能力提出更高門檻。

在實際開發用途中，儘管主流LLM語言理解任務與程式碼生成任務中展現出卓越表現。然而在應用推廣仍面臨顯著挑戰。第一，目前大多數表現最佳的LLM屬於封閉源碼（Closed Source），使用者無法存取模型架構與訓練細節，限制了研究再現性與應用可塑性。第二，這些模型高度依賴高效能圖形處理器（GPU）進行推論，造成「可及性」與「運行成本」的雙重障礙，特別對個人開發者與資源有限的小型機構影響深遠。高階GPU設備價格昂貴，且通常不隨一般消費級電腦提供，使得高效運行這些模型的能力不具普遍性。此外，建置與維護這類運算基礎設施所需經費龐大，可能迫使機構調整預算配置，進而排擠其他關鍵研究與開發活動，削弱其在AI創新生態系中的參與度與競爭力。此外，即便部分LLM以開放介面提供免費存取，使用過程中仍可能牽涉資料上傳與雲端處理，導致潛在的隱私與資安風險（Wu et al., 2023）。特別在開發含有敏感資訊的應用場景中，缺乏對模型內部運作與數據處理流程的透明控制，進一步提高資料外洩的憂慮。

為解決上述問題，研究逐漸將焦點轉向參數較少、運算效率更高的輕量級模型。儘管如 PaLM（Chowdhery et al., 2023）、GPT-3（Brown et al., 2020）與 GPT-4（OpenAI, 2023）等代表性模型分別擁有 1750 億、5400 億至高達 1.7 兆個參數，顯示其龐大的語言建模能力，但其龐大模型規模亦導致部署成本與計算資源需求相對提高。相對地，新一代小型模型如 Mistral（Jiang et al., 2024）與 LLaMA（Dubey et al., 2024）僅以約 70 億參數，在多項任務上展現出與大型模型相當，甚至在特定應用情境中更具優勢的效能表現。這類模型在推論效率與硬體資源需求間取得更佳平衡，為低資源場景下的應用提供可行解方。此外，量化技術的興起也有助於加速模型推論。透過將模型權重轉換為低精度格式，大幅降低記憶體與運算需求的同時，盡可能保留下游任務中的效能表現。例如 Activation-aware Weight Quantization（AWQ）（Lin et al., 2024）與 Generative Pre-trained Transformers Quantization（GPTQ）（Frantar et al., 2022）等方法，已被廣泛應用於壓縮 LLM，並在多項任務中維持接近原始模型的準確度。而為了支援量化模型於低資源環境中部署，工具函式庫如 LlamaCPP（Khosravi, 2024）提供了在 CPU 上執行已量化模型的能力。LlamaCPP 為一種高效的二進位儲存與推論格式，專為 GGUF（GPT-Generated Unified Format）檔案設計，優化了模型載入速度與記憶體使用。使得 LLM 得以在無需高階 GPU 的情況下運行，進一步降低技術進入門檻，擴展其在各類邊緣設備與本地環境中的應用潛能。

除了模型架構與推論效率的優化，提示工程（Prompt Engineering）的發展亦成為提升 LLM 性能的關鍵策略之一。透過精確設計的提示語句，可引導模型生成更具語境關聯與任務對應性的回應（Alier et al., 2024）。提示工程使使用者無需對模型進行再訓練，便能以低成本激發其潛在知識與推理能力，進而擴展模型的應用範疇。具代表性的策略包括 zero-shot 與 few-shot。zero-shot 允許模型僅根據任務描述即執行新任務，few-shot 則藉由提供極少數範例，強化模型對任務模式的理解與應答準確性（Brown et al., 2020）。Chain-of-Thought（CoT）則是將複雜任務拆解為一系列邏輯步驟，逐步引導模型進行中介推理，有效提升其在數學推論、邏輯判斷等高階認知任務中的表現（Zhang et al., 2022）。透過這類提示設計策略，模型不僅能展現出更強的泛化能力，也進一步降低應用門檻，拓寬 LLM 於教育、程式輔助與語意理解等多元場景中的應用潛力。

寫引言節尾跟研究問題

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：相較於傳統分類模型（如 SVM、Adaboost、Bagging），本研究提出之 TIPS Framework 是否能透過提示設計與 LLM 整合，有效提升 Pthread 程式碼分類與生成任務的整體表現？

RQ2：本研究提出之 TIPS Framework在面對任務不均衡與程式樣本多樣性時，是否仍能維持穩定的推論品質與生成效率？

# Related Work：

## LLM 在軟體工程與平行程式設計中的應用

強調LLM於軟體工程與平行程式的**任務導向應用性**，聚焦其在真實開發與並行環境中的效能與挑戰

* LLM於軟體工程任務的應用（程式修復、摘要、API理解）
* LLM於高效能運算（HPC）與平行程式碼的研究應用（如OpenMP pragma生成、平行化潛力分析）
* 專案脈絡下的少樣本提示選擇、錯誤修復任務
* LLM輔助平行化、錯誤偵測、程式碼建議（含ChatGPT、GitHub Copilot）

當前大型語言模型（Large Language Models, LLMs）於軟體工程與平行程式設計領域的應用，逐步展現出跨任務整合與語意推理的突破潛力。此一研究方向不再侷限於自然語言處理，而是深入滲透至真實開發場景與高效能計算（High-Performance Computing, HPC）任務中。其能力涵蓋從程式修復、API 理解到平行化程式碼生成與錯誤偵測等複雜任務，成為未來智慧軟體開發的重要支柱。

在軟體工程領域，LLM 的應用涵蓋理解、生成與修復三大面向。在理解與摘要生成方面，Yang et al., (2023)開發出APICKnow 模型，透過微調大模型識別 Stack Overflow 討論中的 API 實體與語意關係，即使在缺乏大量標註資料的條件下，亦能展現優異性能。Huang et al., (2022)以 code-masked 語言模型處理程式中 fully qualified name（FQN）模糊問題，改善模糊代碼片段的 API 名稱解析效果。這些研究揭示了 LLM 對於程式碼語意的深層理解能力，尤其是在與自然語言融合的程式任務中表現突出。

生成程式任務是LLM另一大應用焦點。Cao et al., (2025)透過系統性實驗評估ChatGPT在多語言程式修復上的泛化能力，發現其在多樣錯誤類型與語法結構下皆具良好適應性，特別是在低資源語言與非典型語法中仍可達成高準確率。Nashid et al., (2023)真對少樣本問題提出了一種基於檢索的提示選擇策略，有效提升了 LLM 在程式修復與測試斷言生成上的表現。Ahmed & Devanbu(2023)則在LLM效能受限於專案脈絡缺失下，透過加入專案內的範例提示進行微調，有助於提升生成摘要的相關性與內容深度。這些研究不僅證明 LLM 可被視為靜態分析工具的補充，更顯示其可在軟體工程流程中扮演具備語境感知能力的輔助開發者。

在高效能運算與平行程式設計領域上，傳統HPC須仰賴專業開發人員針對程式架構進行手動優化與平行化處理。而近期研究則嘗試將LLM導入以簡化開發流程與擴展應用效能。例如Chen et al., (2023)開發的LM4HPC模型，即著眼於透過 LLM 執行程式碼生成與效能最佳化工作。Ding et al., (2023)則以HPC領域問答資料集微調 LLaMA，推出 HPC-GPT，成功將語言模型引導至專業科學計算語境中，使其可協助回答與生成 HPC 程式碼。Kadosh et al., (2023)提出以源碼圖表示與 Transformer 模型相結合的架構，能預測 OpenMP 平行迴圈中的 pragma 語句，從而自動識別平行化潛力。Chen et al., (2024)設計的 OMPGPT 模型，專門針對 OpenMP 程式碼優化進行訓練，可主動產生 pragma 建議，展示領域特定語言模型在高效能程式生成上的強大能力。這些發展指出，語言模型正由單純生成工具轉化為具備編譯導向知識與結構理解的專業系統。

不僅是新模型的設計與應用，ChatGPT 與 GitHub Copilot等現有通用工具亦被證實在高效能運算與平行程式設計領域上具有實際貢獻價值。Mišić & Dodović (2024)分析Copilot 在平行程式碼生成任務中的協助潛能，尤其是在語法提示與模板填寫任務中能大幅降低開發門檻。Alsofyani & Wang(2024)則利用 LLM 偵測 OpenMP 並行錯誤，顯示語言模型亦可支援錯誤辨識任務並提供潛在修正策略。即便非為專業 HPC 訓練的 LLM，經過適當提示設計與任務引導，仍能有效參與平行程式開發過程。

在實作層面，平行程式設計常伴隨如資料競爭（data race）問題，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。而研究則提出靜態分析(static analysis)、動態分析(dynamic analysis)與執行階段驗證(runtime verification)等多種策略。例如Serebryany et al., (2011)於 LLVM 編譯器中整合 ThreadSanitizer，實現動態競爭偵測。Liew et al., (2024針對 GPU 程式引入結合靜態分析與領域特定抽象的技術，強化錯誤捕捉準確性。Choi et al., (2022) 則提出一套混合靜態與動態分析架構，增強對物件導向程式中競爭情境的辨識能力。Malakar et al., (2024)研發的RaceFixer能在偵測錯誤後提供對應的同步策略建議，如自動插入互斥鎖或條件變數以修正問題。儘管這些方法在錯誤偵測與修復上已有顯著成效，然而其多數需依賴複雜的語法與程式控制流程分析，且不易適用於不同語言與架構間的遷移任務。而LLM所提供語意對齊、語境推理與多範式語言適應性，具有潛力補足傳統工具不足之處。特別是在面對結構模糊、語法不一致或不完整的輸入時，LLM 所展現的彈性與修補能力已成為研究者關注焦點。不僅止於支援程式碼生成與理解的語言模型，而是逐步進化為能處理高結構性與領域特定問題的智慧引擎。

整體而言，LLM在HPC與平行設計中協助pragma 生成、平行潛力辨識與錯誤偵測修正。透過適當的微調與提示設計，LLM可達成跨語言、跨專案、跨任務的泛化學習，為軟體開發流程中的關鍵決策提供支持，成為促進程式設計自動化、強化並行任務可控性與推動開發者培育的關鍵核心。

## LLM 生成程式碼的品質與效率問題

此節可強調您研究動機中的\*\*「效率瓶頸與挑戰」\*\*，為後續研究目的鋪陳基礎。

* 稀缺資源場景（IoT、邊緣裝置）下的挑戰，LLM常見的效率低落原因（如 token 預測傾向局部補全、prompt 含糊造成幻覺）
* 程式碼正確性、效率、記憶體與運算資源的評估指標
* GPT與人類程式設計者的效能比較（正確性 vs. 效率）
* 基準數據集（HumanEval、MBPP、EFFIBENCH）的評估方式（含 pass@k）

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)。此外，完整的參數微調需要消耗大量資料與運算資源，也有可能損害模型的泛化能力(Han et al., 2024)。雖然部分參數高效微調(Parameter-Efficient Fine-Tuning, PEFT) 能緩解此問題，但也往往導致災難性遺忘（catastrophic forgetting）(Han et al., 2024)。因此，在程式碼生成任務中，必須採用自訂化提示工程來兼顧結果的準確性與可解釋性。

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。

## 小型與開源 LLM 模型及提示工程的發展

此節不僅回應您研究中關於模型選型與提示策略的設計，也強調小模型+高效率提示的**可行技術解法**。

* 開源與小型LLM（LLaMA、Mistral、Phi、Mixtral、Zephyr、MiniCPM）的表現與架構優勢
* FlashAttention、GQA、SMoE等技術
* Prompt Engineering：zero-shot、few-shot、CoT、Auto-CoT、Self-consistency、LogiCoT
* 提示框架（LangChain、Semantic Kernel）與外部系統整合性
* 提示優化對效率與準確性的貢獻

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)。

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。

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)。

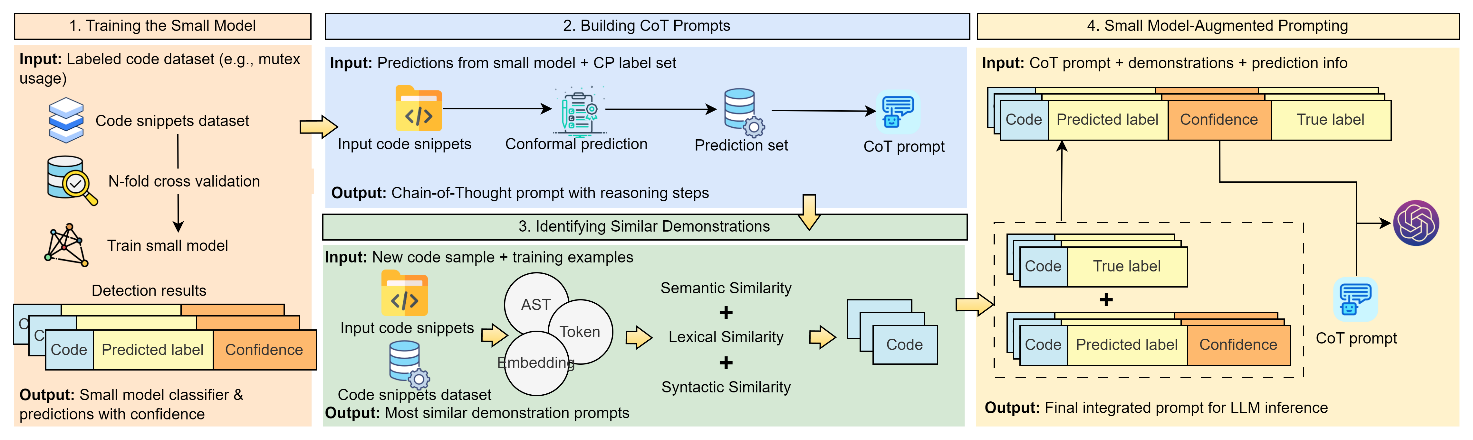
在自動化流程上，Zhang et al., (2022) 提出了「自動化 Chain-of-Thought(Auto-CoT)提示」，能自動產生多樣化的推理鏈，在無需人工設計的情況下提升準確度與穩定性。Wang et al., (2022) 在 CoT 提示的基礎上提出「自我一致性(Self-Consistency)」機制，透過產生多條推理路徑，並選出最一致的答案，藉此進一步提升解題準確性與穩健性。Zhao et al., (2023) 提出了「邏輯推理提示(LogiCoT)」，將符號邏輯原則引入推理過程，用於驗證每個推理步驟的正確性，以降低錯誤與幻覺(hallucination)的發生。

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)。

# Method

本研究系統架構如圖OO所示。系統架構涵蓋四個階段，training the small model、building CoT prompts、identifying similar demonstrations與small model-augmented prompting。系統核心在於利用(In-Context Learning, ICL)與(Chain-of-Thought, CoT)技術來設計高品質提示語，藉此提升LLM在程式生成任務中的表現。

Figure ：An overview of the proposed TIPS Framework



## Benchmark

在本研究中，我們從GitHub上的開源專案中擷取Pthreads原始碼檔案[[1]](#footnote-1)，並特別聚焦於 mutex\_lock 與 mutex\_unlock 的同步模式，以分析這些基本的並行實作實務。我們將這份基準資料集分為兩個子集合，每筆樣本大約包含 50 到 250 行的程式碼。

## Training The Small Model

本研究首先使用一個小型模型作為 LLM 的前端外掛模組，用來強化ICL的構建。並有效激發 LLM 隱含的微調能力。這種策略能夠減輕LLM在偵測任務中常見的幻覺（hallucination）、資料分佈差異（data distribution discrepancies）等效能退化問題。我們首先準備一個Pthreads相關的程式碼片段資料集，並定義為：

其中，為樣本筆數，表示一個樣本的原始程式碼（輸入特徵）。 ∈ {0,1,2,…,n}表示對應的真實標籤，接著，我們定義小型模型為：

其中，是模型對預測出的類別標籤， ∈ [0, 1]為模型對預測結果的信心值（confidence）。而訓練的目標是最小化損失函數，Equation如下：

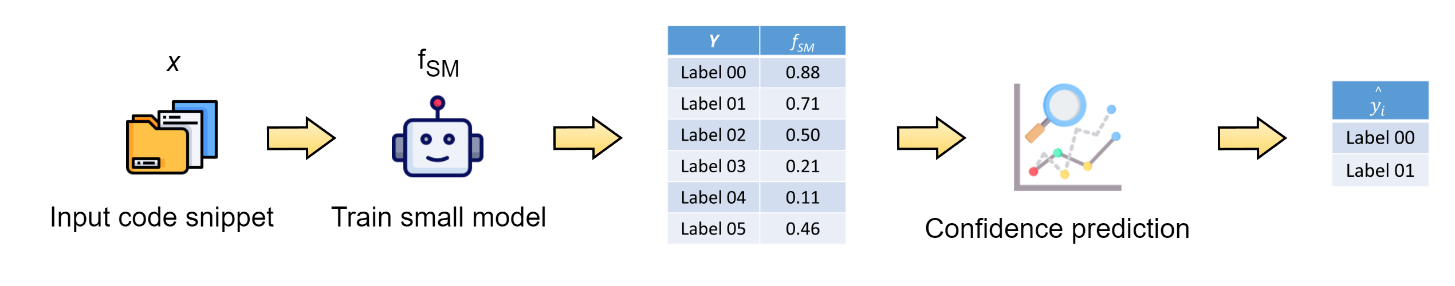
其中，為預測標籤與真實標籤之間的損失。透過這樣的小型模型訓練流程，我們能夠見一個對ICL具備辨識能力的模型，並將其結果作為後續建構 CoT 提示語與支援 LLM 推論的依據。

## Buidling CoT Prompts

系統的第二階段是針對每個輸入的程式碼片段，產生一條特定的推理路徑（reasoning path），這個過程稱為 Chain-of-Thought (CoT)。CoT與傳統僅使用 “input”、”output”的提示不同，CoT會在提示語中加入一系列中間的推理步驟，模擬人類逐步解題的過程。這種策略將複雜問題拆解成一連串較簡單的小問題，使LLM能更聚焦地解決每一步，進而提升整體表現。CoT的構建可以分為兩個階段。第一，使用已訓練的小模型對測試樣本進行預測。第二，套用「符合性預測方法（Conformal Prediction, CP）」產生具有信心保證的候選標籤集合(如Equation(XX)所表示)。

CP的核心是：量化模型預測的不確定性，並為每個預測結果提供信心水準的保證。其中，是測試樣本的預測標籤集合，是一個可能的預測標籤，為非符合性分數，衡量與之間不一致的程度。具體而言，CP會定義一個非符合性函數（non-conformity function），用來衡量模型的預測結果與實際觀察值之間的差異程度。這個非符合性分數（non-conformity score），可以視為「預測不一致程度」的量化指標。之後再透過校準集（calibration set）對此函數進行校準，再將其應用到測試樣本上。透過這樣的方式，LLM 不僅能參考小模型給出的推理步驟與候選標籤，還能透過 CP 機制確保推論結果的可信度，進一步提升整體判斷品質與準確性。

Figure ：An example of CP process



## Generalized CoT prompt template

在建立CoT prompt template上，我們根據現有的 CoT 技術相關研究成果來設計。CoT prompt templatedk包含以下五個推理步驟：

* Semantic Understanding(語意理解): 分析輸入程式碼片段的核心行為與設計意圖，理解它想達成的功能。
* Structural Analysis(結構分析): 檢視程式碼中關鍵的語句與結構元素，辨識可能與某個 ATs 有關的實作組件或模式。
* Classification Judgment(分類判斷): 根據語意與結構分析的結果，判斷該程式碼片段是否與某一個 AT 標籤有關。
* Label Identification(標籤辨識): 若程式碼與某個 AT 有關，則從 CP 所提供的預測標籤集合中選出最符合的標籤；若無關，則判斷是否屬於預測集合之外的其他類別。
* CoT Construction(CoT 推理整合): 將上述所有推理步驟有系統地整合為一個連貫的思考過程，以建立針對該偵測樣本的 CoT 提示語，協助 LLM 更準確地做出判斷與標註。

我們將CoT 模板與 CP 結果結合，為每個待偵測的樣本產生客製化的 CoT 提示語。舉例而言，若某份輸入程式碼樣本被小型模型初步偵測為” Mutex-based strategy for critical section protection.”，而符合性預測（CP）回傳的候選標籤集合為{critical\_section, resource\_pooling}，這個集合以 95% 的信心水準涵蓋了正確標籤。此時系統就會依據：

* 小型模型的預測結果: ” Mutex-based strategy for critical section protection.”
* CP 給出的候選集合: {critical\_section, resource\_pooling}
* CoT 推理模板(Semantic Understanding → Structural Analysis → Classification Judgment → Label Identification → CoT Construction)

最終產生一組針對該 Pthreads 程式碼片段的客製化 CoT 提示語，協助 LLM 正確判斷其同步策略的意圖與類別，並能生成具備 pthread\_mutex\_lock() / unlock() 正確結構的並行程式碼。

## Identifying Similar Demonstrations

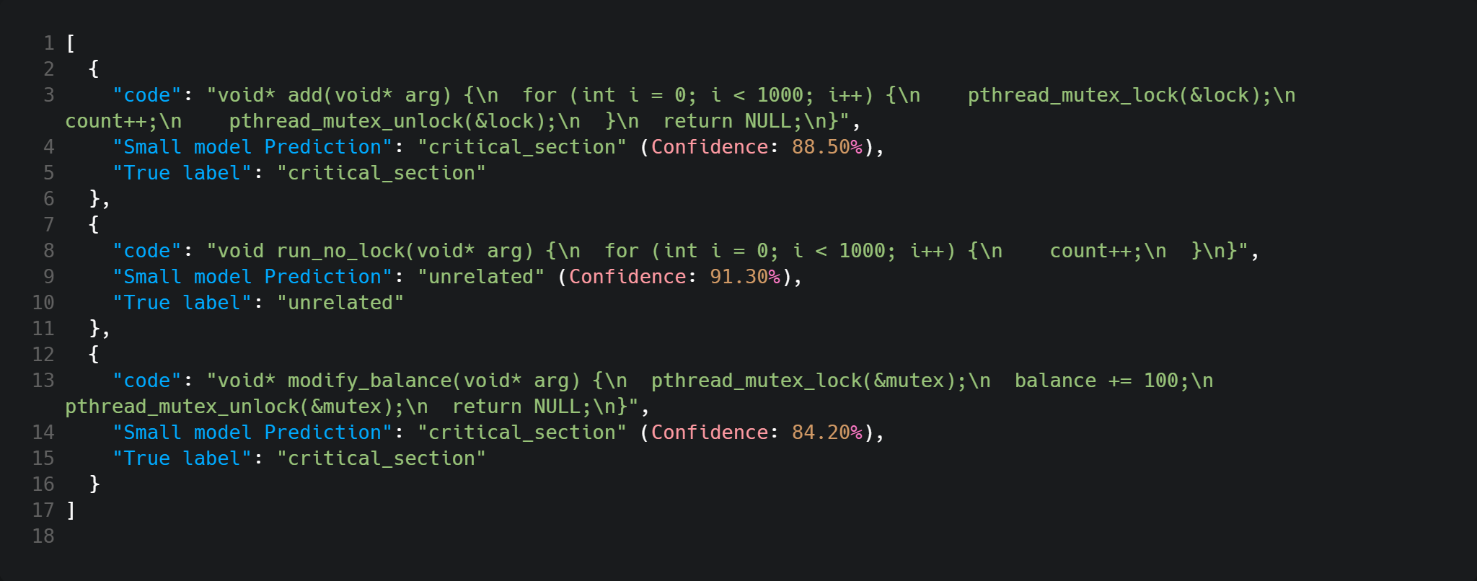
在LLM應用上，In-Context Learning(ICL)是一種常見的使用方式。而如何設計有效的demonstrations則是一個關鍵的研究問題(Zhao et al., 2023)。透過ICL提供的範例，LLM能夠從中生成與任務相關的元梯度（meta-gradient），並透過注意力機制隱式地執行梯度下降，進而達到類似微調（fine-tuning）的效果(Von Oswald et al., 2023)。若我們定義示範集合(demonstration set)為：

其中，為將訓練資料轉換為自然語言提示的函數。而ICL的的推理流程可表示為：

其中，為待預測樣本，為LLM 輸出的預測結果，為任務描述(task instruction)，為示範集合(demonstration set)。此外，為了程式碼生生任務中ICL的效果，本系統會同時考慮語意相似度（Semantic Similarity）、字面相似度（Lexical Similarity）與語法相似度（Syntactic Similarity）來找出與當前輸入樣本最相似的範例。語意相似度（Semantic Similarity）如Equation(XX)所表示。用於計算兩個嵌入向量的餘弦相似度，其中分別為當前與示範程式碼的語意向量。字面相似度（Lexical Similarity）如Equation(XX)所表示，我們使用Jaccard 相似度計算兩組 token 集合的交集與聯集比例。語法相似度（Syntactic Similarity）則如Equation(XX)所表示。我們將AST 轉為節點序列，再用 Levenshtein 距離計算相似程度。

最後，我們整合上述三項相似度作為總體相似分數，如Equation(XX)所表示。其中，因為我們將三者視為同等重要（平均權重）。接著，我們從範例集合中選出與當前樣本相似度最高的程式碼，並以該程式作為 ICL 的示範(如Figure 2所表示)。

Figure ：An example of demonstrations in ICL prompt



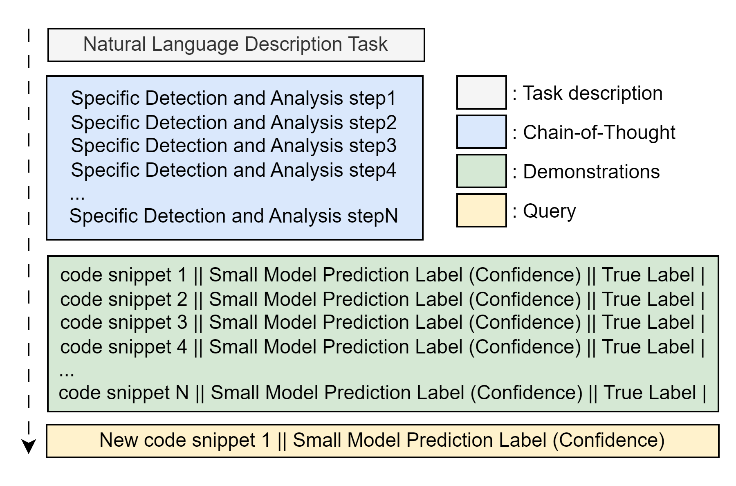
## Model-augmented Prompting

當我們完成 Chain-of-Thought（CoT）提示語的客製化，以及 In-Context Learning（ICL）示範樣本的選取與整合後，便可以將上述步驟進一步整合為一個完整的提示語設計（prompt design），用於程式碼生成任務。本系統所提出的提示語框架包含兩個主要元件：

* Chain-of-Thought Prompts：支援模型從複雜問題中拆解出「子任務」，幫助模型建立明確的推理過程，提升預測透明度。
* 小模型預測結果與信心分數（Prediction + Confidence）：協助 LLM 理解樣本程式碼與小模型預測之間得對應關係。

最終prompt如Figure 3所示，藉由這種動態決策機制，模型可以根據情況選擇「是否信任小模型的預測」，或是「依靠自己的推理結果」做出最終分類。這種設計不僅讓 LLM 在面對模糊或多義的程式碼時更具彈性，亦能有效提升其在策略性架構偵測任務中的整體表現與解釋能力。

Figure ：The final prompts template



# Experimental Design

為了評估TIPS Framework在生成Pthread程式碼任務的準確性表現，我們設計了多組比較實驗，並選定Support Vector Machine (SVM)(Joachims, 1999)、Adaboost(Freund & Schapire, 1996)、Bagging(Breiman, 1996)作為對照組比較。這些模型涵蓋了傳統分類器與深度語言模型方法，構成多樣化比較基準，藉此全面評估TIPS Framework的生成效能。

# Result

Table ：Performance Comparison of Small Models and TIPS Framework in Pthread Code Generation Tasks (Balanced Dataset)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Task |  | SVM |  |  | Adaboost |  |  | Bagging |  |  |  | TIPS Framework |
|  | Precision | Recall | F1 Score | Precision | Recall | F1 Score | Precision | Recall | F1 Score | Precision | Recall | F1 Score |
| Thread Lifecycle | 0.91 | 0.40 | 0.57 | 0.88 | 0.70 | 0.78 | 0.70 | 0.78 | 0.88 | 1.00 | 0.89 | 0.89 |
| Mutex Management | 0.71 | 0.80 | 0.89 | 0.89 | 0.73 | 0.80 | 0.80 | 0.80 | 0.80 | 1.00 | 0.91 | 0.95 |
| Condition Variables | 0.63 | 0.60 | 0.75 | 0.88 | 0.80 | 0.78 | 0.88 | 0.80 | 0.80 | 1.00 | 0.91 | 0.95 |
| Thread Pool Creation | 0.83 | 0.70 | 0.82 | 0.91 | 1.00 | 1.00 | 0.95 | 1.00 | 0.95 | 0.95 | 0.91 | 0.93 |
| Scheduling Policy Assignment | 0.91 | 0.66 | 0.89 | 0.89 | 0.88 | 1.00 | 0.78 | 1.00 | 1.00 | 0.92 | 0.88 | 0.96 |
| **Average** |  |  | 0.78 |  |  | 0.87 |  |  | 0.89 |  |  | 0.96 |

Table ：Robustness Evaluation of Small Models and TIPS Framework under Imbalanced Pthread Task Distribution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | SVM | Adaboost | Bagging | TIPS Framework |
|  | F1 Score | F1 Score | F1 Score | F1 Score |
| Thread Lifecycle | 0.17 | 0.20 | 0.33 | **0.22** |
| Mutex Management | 0.55 | 0.52 | 0.63 | **0.76** |
| Condition Variables | 0.00 | 0.49 | 0.64 | **0.60** |
| Thread Pool Creation | 0.05 | **0.67** | 0.51 | 0.44 |
| Scheduling Policy Assignment | 0.63 | 0.49 | 0.84 | **0.72** |
| **Average** | 0.32 | 0.45 | 0.60 | **0.55** |

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