



Society Section: IEEE Education Society Section

TIPS: A Prompt Engineering Framework for Code Classification and Generation on Resource-Constrained Systems

Submission ID 82950bb2-0bba-4c62-b8d9-abeff464289c

Submission Version Initial Submission

PDF Generation 07 Aug 2025 10:47:59 EST by Atypon ReX


Authors

Mr. WEI-CHENG CHEN

Affiliations

- Department of Engineering Science, National Cheng Kung University, Tainan 70101, Taiwan

Prof. SHIH-YHE CHEN
Corresponding Author
Submitting Author

 [ORCID](https://orcid.org/0000-0002-1863-6168)
<https://orcid.org/0000-0002-1863-6168>

Affiliations

- Department of Engineering Science, National Cheng Kung University, Tainan 70101, Taiwan

Additional Information

Subject Category

Computational and artificial intelligence
Computers and information processing
Education

Keywords

Computational and artificial intelligence
Computer science education
Engineering education
IEEE Education Society
IEEE Learning Technology

IEEE Transactions on Education
Programming environments
Select a Manuscript Type
Research Article

Files for peer review

All files submitted by the author for peer review are listed below. Files that could not be converted to PDF are indicated; reviewers are able to access them online.

| Name | Type of File | Size | Page |
|-----------------|---------------------|--------|------------------------|
| TIPS_0807V1.pdf | Main Document - PDF | 1.4 MB | Page 4 |

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

TIPS: A Prompt Engineering Framework for Code Classification and Generation on Resource-Constrained Systems

WEI-CHENG CHEN, and, SHIH-YHE CHEN

Department of Engineering Science, National Cheng Kung University, Tainan 70101, Taiwan

Corresponding author: Shih-Yhe Chen (e-mail: sychen-ncku@gs.ncku.edu.tw).

ABSTRACT Recent advances in AI-assisted programming have highlighted the potential of large language models (LLMs) to perform complex code classification and generation tasks. However, their substantial computational demands and reliance on large-scale datasets limit their applicability in real-world environments with constrained platforms such as educational tools, edge devices, and small development environments. To overcome these challenges, this paper proposes the Template-Integrated Prompting System (TIPS) framework, a modular and efficient prompt engineering architecture designed specifically for resource-limited scenarios. TIPS framework combines few-shot learning with Chain-of-Thought (CoT) reasoning, enabling lightweight models to approximate the performance of LLMs in code understanding and generation. The TIPS framework integrates confidence-based small model training, semantically guided demonstration selection, and structured prompt templates to construct an interpretable and adaptable reasoning pipeline. Experiments conducted on a benchmark dataset of multithreaded Pthread-based programs demonstrate that the TIPS framework achieves significant improvements over traditional classifiers and prompt-based methods in terms of precision, recall, and F1-score, even under class imbalance conditions. The results suggest that the effectiveness of the model is influenced not only by scale but also by the quality of semantic guidance and the reasoning structure. The proposed framework provides a scalable and interpretable solution for advancing AI-assisted programming in resource-constrained environments.

INDEX TERMS Prompt Engineering, Multithreaded Programming, In-Context Learning, Chain-of-Thought, Large Language Models, Code Generation

I. INTRODUCTION

Code generation has emerged as a key area of research and practical interest, driven by rapid advances in large language models (LLMs) and natural language processing (NLP) [1]. LLMs such as GPT-4, Claude, and Gemini demonstrate strong abilities to generate code from NLP. This capability is significantly reshaping conventional software development workflows. LLMs enable the transformation of natural language into structured code, thus simplifying programming processes and accelerating the software development lifecycle [2]. This popularity is largely attributed to their versatility across diverse domains and their ability to offer context-aware real-time assistance. This enables developers to receive real-time assistance during the coding process, which is particularly beneficial in autonomous learning and self-directed programming environments [3]. LLMs address diverse learning needs by providing structured guidance, semantic summation, and natural language explanations. These features con-

tribute to greater clarity of instruction and improved learning outcomes. Beyond programming tasks, LLMs can generate context-aware content to support writing assistance, educational material development, and AI-driven programming education [4].

LLMs can understand program semantics, enabling the generation of syntactically correct and functionally valid code [5] [6] [7]. Empirical findings indicate that pre-trained models perform well in program comprehension, supporting tasks such as full-function generation, code completion, and fine-grained correction at the line level. LLMs have demonstrated robust generalization across programming languages (e.g., Python, Java) and a variety of development settings [6]. Cao et al. [8] further investigated the role of prompt engineering in improving LLM-based debugging. They introduced tasks involving structurally complex bugs with high interdependencies to assess the practical utility of LLMs in realistic debugging contexts.

The performance of LLMs in code generation remains inferior to their performance in natural language generation tasks [9]. First, code generation must ensure an accurate interpretation of natural language instructions while adhering to strict syntactic and semantic constraints of programming languages [10]. Minor syntax errors can result in compilation failures (e.g., due to missing symbols) or cause unintended logic errors at runtime. These challenges underscore the growing importance of precise language comprehension and logical reasoning in code-related tasks. Second, many leading LLMs are closed-source, limiting access to model architectures and training data. This lack of transparency hinders reproducibility and makes customization difficult. Third, current LLMs rely on GPU-based inference, posing significant barriers for individual developers and small-scale deployments. High-performance GPUs are costly and not universally available. Furthermore, public LLM services often require cloud-based execution, raising privacy concerns due to limited transparency in internal data handling and processing workflows [11].

Motivated by these constraints, recent research has increasingly focused on lightweight models with reduced parameter sizes and lower computational overhead. LLMs such as PaLM (175B) [12], GPT-3 (175B) [13], and GPT-4 (up to 1.7T) [14] have shown strong abilities in language understanding and generation. However, the sheer scale of these models incurs significant computational and deployment costs, which restricts their practical applicability. In contrast, more compact models such as Mistral and LLaMA (each with approximately 7B parameters) have been shown to match or even outperform LLMs in specific tasks [15] [16]. These models strike a better balance between inference speed and hardware requirements, making them suitable for deployment in resource-constrained settings.

In addition, quantization has accelerated inference by converting model weights into low-precision formats, significantly reducing memory and computational demands while generation quality. Techniques such as Activation-aware Weight Quantization (AWQ) [17] and Generative Pre-trained Transformers Quantization (GPTQ) [18] have been widely adopted in recent work for model compression, as they maintain accuracy comparable to full-precision models across a range of tasks. To enable deployment in low-resource or offline environments, tools such as LlamaCPP [19] support CPU-based execution of quantized models. LlamaCPP is designed for the GPT-Generated Unified Format (GGUF), a binary standard that improves model loading speed and reduces memory consumption. This makes LLM deployment feasible without high-end GPUs, expanding accessibility for edge computing and on-device applications.

Although lightweight models and quantization improve inference efficiency and portability, they remain insufficient for code generation tasks, which demand high structural precision, semantic depth, and minimal tolerance for errors [20]. Prompt engineering has proven to be an effective strategy to minimize these challenges. Without retraining, users can

elicit latent knowledge and reasoning capabilities from the model without retraining, thus expanding its applicability at minimal computational cost [21] [22]. Notable prompting strategies include zero-shot [23], few-shot [24], and CoT prompting [25]. Zero-shot enables the model to handle novel tasks given only the task description. Few-shot enhances performance by providing the model with a small set of annotated examples. CoT explicitly structures the reasoning process by decomposing complex tasks into sequential steps, improving performance in multistep inference tasks such as logical deduction and mathematical reasoning. Well-designed prompts have been shown to significantly improve performance in code completion, error correction, and template-based code generation, particularly compared to baseline prompting approaches [26]. When combined with a few-shot or CoT techniques, prompt engineering enhances semantic alignment, generalization, while facilitating broader deployment across domains such as education, development tools, and semantic analysis.

This study introduces the Template-Integrated Prompting System (TIPS) framework that enables small models to classify and generate code under resource constraints. The TIPS framework combines In-Context Learning (ICL) and CoT reasoning to design structured prompt templates that guide LLMs in generating semantically coherent and logically consistent output. We evaluated the framework for parallel programming tasks, such as managing critical sections, handling condition variables, and selecting scheduling strategies. Empirical results show that TIPS framework effectively generates high-quality code even in semantically complex scenarios, such as concurrent thread synchronization. The main contributions of this paper are summarized as follows:

- TIPS framework is a lightweight prompt engineering framework designed for resource-constrained environments, integrating small models with large-model reasoning via ICL and CoT to enhance classification accuracy and inference consistency.
- Its effectiveness is validated on multithreaded pthread code generation tasks, focusing on synchronization strategy recognition and structured output for educational and development use.
- Ablation studies confirm the contributions of key components, including small models, CoT reasoning, and demonstrations. These components collectively improve interpretability, stability, and extensibility.

II. LITERATURE REVIEW

A. LLMS APPLICATIONS IN PARALLEL DESIGN

LLMs have NLP to support real-world parallel programming tasks in High Performance Computing (HPC). They enable advanced functions such as code repair, API comprehension, multithreaded code generation, and error detection [27]. Yang et al. proposed APICKnow, a fine-tuned model that accurately recognizes API entities and their semantic relations in Stack Overflow discussions [28]. Huang et al. addressed the ambiguity in fully qualified names (FQNs) by using code-masked

language modeling, significantly improving the resolution in incomplete code segments [29]. These works underscore the ability of LLMs to deeply understand programming semantics, especially in tasks that combine natural language and code. Cao et al. evaluated the performance of ChatGPT in multilingual code repair, demonstrating its adaptability to various types of errors and syntactical structures [8]. Nashid et al. introduced a retrieval-based prompt selection mechanism to overcome few-shot limitations, significantly enhancing LLMs performance in code repair and assertion generation [30]. Ahmed and Devanbu demonstrated that project-specific example prompts can fine-tune LLMs outputs under a constrained context, improving relevance and depth [31]. Collectively, these studies show that LLMs not only supplement static analysis tools but also act as context-sensitive assistants within software engineering workflows.

Traditionally, parallel programming has relied on manual efforts by developers to restructure code to optimize performance. A primary challenge in this domain is data races, which arise when multiple threads access shared variables concurrently without proper synchronization, potentially resulting in non-deterministic behavior or program crashes. To mitigate data races, synchronization mechanisms, such as condition variables, mutexes, and semaphores, are commonly employed in parallel programming to maintain thread safety. Previous studies have introduced strategies, such as static analysis, dynamic analysis, and runtime verification, to detect and resolve concurrency-related issues.

Serebryany et al. [32] integrated ThreadSanitizer into the LLVM compiler to enable dynamic analysis. Liew et al. [33] improved the static analysis of GPU programs by incorporating domain-specific abstractions, improving error detection accuracy. Choi et al. [34] proposed a hybrid framework combining static and dynamic analysis to improve the detection of race condition in object-oriented programs. Malakar et al. [35] introduced RaceFixer, which detects synchronization issues and suggests corrective actions, such as inserting condition variables or mutexes. Although these methods demonstrate success in identifying and correcting errors, their dependence on syntactic and control flow analysis limits adaptability across languages and systems. LLMs offer advantages in semantic alignment, contextual reasoning, and language flexibility, by addressing these limitations. Their ability to process ambiguous or incomplete input underscores their potential as intelligent engines for structured domain-specific tasks.

LLMs have recently been applied to streamline development workflows and enhance performance under HPC conditions. Chen et al. [36] introduced LM4HPC, a model designed for semantic understanding and structural adaptation in HPC scenarios, demonstrating robustness in automated design and optimization. Kadosh et al. [37] presented a hybrid framework integrating code graphs with Transformers to predict pragma directives in OpenMP parallel loops, effectively identifying parallelization opportunities. Ding et al. [38] fine-tuned LLaMA on an HPC-specific QA dataset to produce

HPC-GPT, which helps in HPC code generation. Chen et al. [1] proposed OMPGPT, a model trained specifically for OpenMP optimization, capable of suggesting pragma usage and highlighting the strength of domain-specific LLMs in high-performance code generation.

Beyond developing new models, general-purpose tools such as ChatGPT and GitHub Copilot have also shown practical value in parallel programming design. Mišić and Dodović [39] analyzed Copilot's assistance in parallel code generation, particularly in syntax prompting and template-based suggestions, which significantly lower development barriers. Alsofyani and Wang [40] used LLMs to detect concurrency errors in OpenMP programs. Collectively, these studies demonstrate that LLMs, even without specialized HPC training, can achieve cross-language, cross-project, and cross-task generalization when guided by well-designed prompts. Consequently, LLMs support critical decision making in parallel design by enabling code automation, enhancing concurrency control, and facilitating developer education and skill development.

B. ASSESSING THE PRACTICALITY OF LLM CODE GENERATION BASED ON CORRECTNESS AND EFFICIENCY

LLMs demonstrate considerable promise in parallel programming and HPC. However, integrating them into development workflows raises concerns about code quality, resource consumption, and operational efficiency. Unlike traditional tools, LLM-generated code often exhibits non-deterministic behavior, unlike traditional programming tools. Similar issues have been reported on the reliability of predictive outputs in fields such as NLP [41], computer vision [42], autonomous driving [43], and healthcare [44].

Inefficiency of LLM-generated code can slow program execution, increase operational costs, and violate constraints such as memory, latency, or energy efficiency limits. Thus, usability and performance in resource-limited environments must be critically and systematically evaluated [20]. Therefore, both usability and performance under restricted hardware environments must be systematically evaluated [20]. In compute-constrained scenarios such as IoT, edge computing, and cloud devices, memory efficiency and execution latency are of paramount importance [45] [46]. Common evaluation metrics include memory usage, CPU utilization, runtime, and code complexity [47] [48]. Huang et al. [49] introduced the EFFIBENCH benchmark, showing that although LLM-generated code is functionally correct, it consistently demonstrates inferior performance compared to human-written code, particularly with respect to memory consumption and runtime [14].

Although LLMs excel in structured coding tasks, they often struggle to handle ambiguous or under-specified problem descriptions. In contrast, human-authored code consistently demonstrates superior efficiency [50] [51]. Niu et al. [52] evaluated efficiency using both accuracy and execution time in HumanEval, MBPP (Python), and LeetCodeEval (C++). Their findings indicate that the LLM-generated code contin-

ues to perform worse than the human-written code. Du et al. [53] define code efficiency as the ability to perform tasks with minimal resource utilization, such as time, memory, and computational power, thus improving user experience, reducing energy consumption, and improving cost effectiveness.

Finally, mixed-quality training data can induce hallucinations in LLMs, leading to unnecessary energy consumption and poor performance in critical tasks [54]. Although full fine-tuning improves accuracy, it imposes a high computational cost and may reduce generalization capability. Parameter-efficient fine-tuning (PEFT) reduces training cost but risks catastrophic forgetting of previously learned knowledge [55].

C. SEMANTIC-GUIDED PROMPTING AND COMPACT MODEL DEVELOPMENT

Given the trade-offs between execution efficiency and behavioral consistency in LLMs, recent studies have shown a growing emphasis on enhancing the applicability of LLMs to code generation tasks via two main approaches. One strategy involves adopting lightweight open-source models as feasible substitutes for large proprietary systems. The second strategy emphasizes task-aware prompt engineering, customized to match problem domains or code generation complexity.

Open-source models have been increasingly recognized as effective alternatives to proprietary models, demonstrating competitive performance in terms of accuracy and resource efficiency in code generation tasks. InCoder [56] and StarCoder [57], both specially specifically fine-tuned for program synthesis, exemplify the growing trend of open source models that match or exceed proprietary alternatives in code generation. InCoder employs causal masking, which integrates the strengths of both causal and masked language modeling. In contrast, StarCoder incorporates FlashAttention to accelerate attention computation and reduce memory usage, improving overall computational efficiency. Rozière et al. [58] presented Code Llama, a model built on LLaMA-2 and fine-tuned using causal masking and Long Context Fine-Tuning (LCFT), allowing processing of sequences up to 16,384 tokens." Although later replaced by LLaMA-3 [59] and LLaMA-3.1 [16], these newer models benefit from larger scales and improved training data. In terms of performance, although it comprises only 7 billion parameters, Code Llama surpassed the 70 billion-parameter LLaMA model on benchmarks such as HumanEval and MBPP, indicating a superior capability to manage large-scale code generation tasks [60]. Although subsequently superseded by LLaMA-3 [59] and LLaMA-3.1 [16], Code Llama remains notable for its efficiency. The newer models benefit from increased parameter scales and enhanced training datasets, leading to further improvements in performance.

Beyond the ongoing advancement of the LLaMA series, the introduction of Phi models reflects an increasing research focus on parameter-efficient and scaling-optimized model architectures [61]. Phi-1, a 1.3 billion-parameter transformer model, was specifically developed for code generation tasks

and trained on high-quality textbook-style datasets to improve accuracy and consistency. Building on this foundation, Phi-1.5 [22] further refined next-token prediction capabilities and was trained on a significantly expanded corpus comprising several billion tokens. Phi-2 [62], with 2.7 billion parameters, further extended the series by combining Phi-1.5 training data with synthetic NLP datasets and curated web content.

Inspired by instruction-following models such as LLaMA, ChatGPT, Claude, and Microsoft BingChat, Taori et al. [63] presented Alpaca, a 7B-parameter model fine-tuned from LLaMA using 52,000 instruction-following examples. Despite having only 7 billion parameters, Alpaca shows a performance comparable to OpenAI's instruction-tuned model text-davinci-003, particularly in instruction-following tasks. Jiang et al. [15] introduced Mistral, an efficient 7B-parameter Transformer model that incorporates Grouped Query Attention (GQA) [64] for faster inference and Sliding Window Attention (SWA) [65] to effectively handle long sequences. Mistral outperformed LLaMA 2 13B in multiple benchmarks, including code generation, and further demonstrated superior performance compared to LLaMA 2 34B in selected reasoning and generation tasks [66]. Extending this architectural approach, Mixtral 8×7B employed a Sparse Mixture of Experts (SMoE) framework to enhance inference efficiency and responsiveness, delivering performance comparable to GPT-3.5 and surpassing LLaMA-2 70B on benchmarks such as MMLU and HumanEval. Models such as Zephyr [67] and MiniCPM [68] incorporate advanced alignment techniques, including UltraChat, UltraFeedback, and Direct Preference Optimization (DPO), to enhance contextual reasoning and optimize alignment with human preferences. These trends underscore a crucial insight: a model's effectiveness is shifting from sheer parameter count to sophistication in prompt engineering and context modeling.

Prompt engineering was first characterized by zero-shot prompting, in which models are guided to perform tasks solely by natural language instructions, without the use of exemplar demonstrations [23]. The few-shot builds on the zero-shot paradigm by incorporating a small set of input-output exemplars, helping the model acquire the expected task format and underlying logical structures [24]. CoT facilitates intermediate reasoning processes, which improve logical inference in complex multistep tasks such as solving a mathematical problem and answering the compositional question [25]. Niu et al. [52] observed that the use of simple prompts for basic tasks and CoT for complex tasks led to improved performance, reduced execution time by 15%, and decreased memory usage by 12%, suggesting CoT's utility as an effective prompt-based optimization strategy. Zhang et al. [69] proposed Automatic CoT (Auto-CoT), a method that autonomously generates semantically diverse reasoning paths without human intervention. This approach improves both model accuracy and output stability. Wang et al. [70] proposed a self-consistency decoding strategy, which aggregates consensus across multiple reasoning paths to identify the most plausible solution. Zhao et al. [69] proposed a self-

consistency decoding strategy, which aggregates consensus across multiple reasoning paths to identify the most plausible solution.

Recent advances in prompt engineering have led to significant improvements in the accuracy and stability of language models when performing complex tasks. In addition, these techniques have improved controllability, interpretability, and computational efficiency. Despite these advances, incorporating prompt-based optimization techniques into a systematic and task-adaptive framework continues to present significant challenges, particularly in aligning prompts with real-world task semantics. This study proposes the TIPS framework, a modular prompting architecture that integrates small-model assistance, semantic retrieval, CoT reasoning, and stepwise example generation, each contributing to task relevance and logical coherence. The TIPS framework improves the logical consistency of CoT reasoning and facilitates prompt design that mirrors the semantic structure and reasoning flow of real-world code generation tasks. By addressing the lack of modular and reusable components in existing prompting strategies, the TIPS framework yields measurable gains in generation accuracy, interpretability, and computational efficiency, indicating a transition from ad hoc prompt design to structured, task-aware frameworks.

III. THE PROPOSED FRAMEWORK: TIPS FRAMEWORK

The TIPS framework consists of four stages: Training the small model, Building CoT prompt, Identifying similar demonstration, and Small model-augmented prompt. The core objective is to improve the performance of the LLMs in code generation tasks by ICL and CoT techniques to construct high-quality prompts, as illustrated in Figure 1.

A. TRAINING THE SMALL MODEL

Strengthen ICL construction and mitigate subtle limitations in LLMs, such as hallucination and data distribution discrepancies. We first introduce a small model as the front-end module of the TIPS framework. Given a labeled dataset segment D , we define:

$$D = (x_i, y_i)_{i=1}^N \quad (1)$$

where N denotes the number of samples, x_i represents the input characteristics of the sample i -th and $y_i \in \{0, 1, 2, \dots, n\}$ is the corresponding ground truth label. The small model prediction is defined as:

$$f_{SM}(x_i) = (\hat{y}_i, c_i) \quad (2)$$

where \hat{y}_i denotes the predicted label and $c_i \in [0, 1]$ is the confidence score. The model is trained by minimizing the following objective function:

$$L = \frac{1}{N} \sum_{i=1}^N \ell(\hat{y}_i, y_i) \quad (3)$$

where $\ell(\hat{y}_i, y_i)$ is the loss function that measures the discrepancy between the predicted and true labels. This training pipeline enables the development of a lightweight classifier with strong ICL discrimination capabilities. Provides reliable support for CoT-based prompt design and enhances the accuracy of the reasoning downstream LLMs.

B. BUILDING COT PROMPT

The second stage of the TIPS framework involves CoT for each input code segment to guide LLMs in generating more accurate and logically consistent outputs. Unlike traditional prompts composed solely of input-output pairs, CoT introduce intermediate reasoning steps that decompose complex tasks into simpler subproblems. This approach mimics human step-by-step reasoning and enhances the interpretability and quality of the output.

The CoT construction process involves two steps: (1) using the trained small model to generate preliminary predictions along with candidate labels; and (2) incorporating these outputs into a structured CoT template for subsequent reasoning.

- **Structural Analysis:** Analyzes syntax and control flow to identify key code components.
- **Semantic Interpretation:** Infers the roles of code elements based on context and meaning.
- **Classification Judgment:** Predicts the task type or sync strategy from structural and semantic cues.
- **Label Mapping:** Selects the best label and handles uncertainty through refinement.
- **CoT Construction:** Combines reasoning steps into a coherent natural language prompt.

For example, assume that an input code segment is initially classified by the small model as ‘Mutex-based strategy for critical section protection.’ The candidate label set returned by Conformal Prediction (CP) is $\{\text{critical_section}, \text{critical_section}\}$, which covers the correct label with confidence level 95%. In this case, the system proceeds on the following bases:

- The small model’s prediction: “Mutex-based strategy for critical section protection.”
- The CP-derived candidate set: `critical_section`, `resource_pooling`
- The CoT reasoning template involves the following steps: Semantic Understanding, Structural Analysis, Classification Judgment, Label Identification, and finally CoT Construction.

To generate reasoning prompts tailored to each test sample, CoT guidance assists LLMs in producing correct synchronization strategies (e.g., `pthread_mutex_()/unlock()`) during inference, thus enhancing output accuracy and reliability. To ensure prediction confidence during reasoning, CP is employed, as defined in Equation (4).

$$\hat{Y}_i = \left\{ y^{(j)} \mid \text{inconf}_i^{(j)} \leq \text{quantile}_{1-\alpha} \left(\left\{ \text{inconf}_i^{(k)} \right\}_{k=1}^m \right) \right\} \quad (4)$$

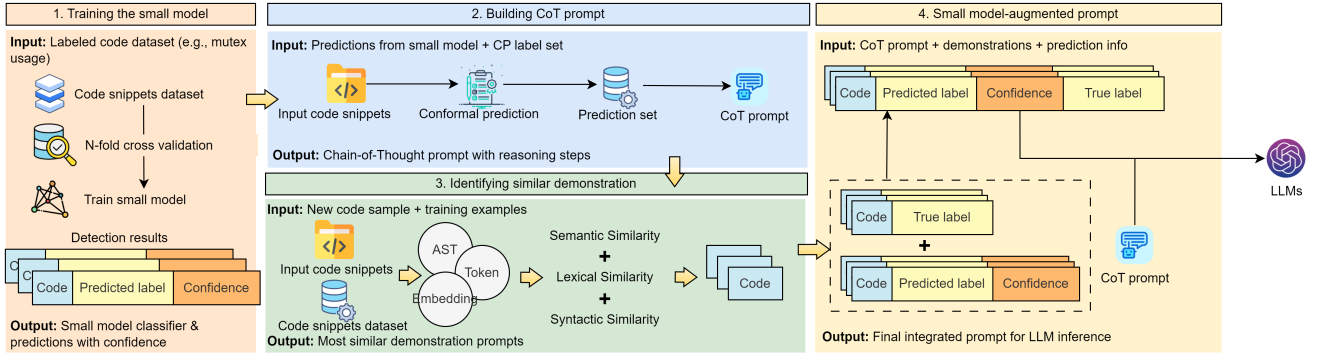


FIGURE 1. An overview of the proposed TIPS Framework.

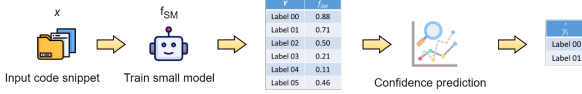


FIGURE 2. Overview of the CP process.

Here, \hat{Y}_i denotes the set of candidate labels for the test input x_i , $y^{(j)}$ is a potential label, and $\text{inconf}_i^{(j)}$ is the non-conformity score, measuring the deviation between $y^{(j)}$ and x_i . CP aims to quantify prediction uncertainty and provide statistical confidence guarantees. Specifically, it defines a non-conformity function to evaluate the mismatch between model predictions and actual observations. This nonconformity score is calibrated in a validation set before applying CP to unseen samples. A typical CP pipeline is illustrated in Figure 2.

C. IDENTIFYING SIMILAR DEMONSTRATION

In prompt engineering, ICL is a widely adopted and effective strategy. Designing high-quality demonstrations remains a core research challenge. By providing tasks-relevant demonstrations, LLMs can implicitly acquire gradient directions and perform adaptation processes similar to fine-tuning through attention mechanisms [71]. Hence, improve reasoning on downstream tasks, thereby improving reasoning on downstream tasks. The demonstration set is defined as:

$$\text{Demo}_i = f(x_1, y_1), \dots, f(x_i, y_i) \quad (5)$$

where $f(x_i, y_i)$ is a function that converts training samples into natural language prompts.

The ICL reasoning process can be expressed as:

$$F_{\text{LLM}}(I, \text{Demo}_i, f(x_{i+1})) = \hat{y}_{i+1} \quad (6)$$

where I is the task instruction, Demo_i denotes the demonstration set, x_{i+1} is the input instance to be predicted, and \hat{y}_{i+1} is the output predicted by the LLM.

To enhance the effectiveness of ICL in code generation, the proposed system identifies the most relevant demonstrations

by jointly considering semantic, lexical, and syntactic similarity. The semantic similarity is computed as follows:

$$\text{Semantic_similarity}(\vec{v}_c, \vec{v}_i) = \frac{\vec{v}_c \cdot \vec{v}_i}{\|\vec{v}_c\| \cdot \|\vec{v}_i\|} \quad (7)$$

where \vec{v}_c and \vec{v}_i are the vector embeddings of the current and candidate code samples, respectively. The lexical similarity is defined as:

$$\text{Lexical_similarity}(T_c, T_i) = \frac{|T_c \cap T_i|}{|T_c \cup T_i|} \quad (8)$$

where T_c and T_i are the sets of code tokens, measured using the Jaccard similarity. The syntactic similarity is computed by comparing abstract syntax trees (ASTs):

$$\text{Syntactic_similarity}(S_c, S_i) = 1 - \frac{\text{Lev}(S_c, S_i)}{\max(\text{len}(S_c), \text{len}(S_i))} \quad (9)$$

where S_c and S_i are sequences of the AST-derived node, and Lev denotes the Levenshtein distance. The overall similarity score aggregates the three components using weighted summation.

$$\text{Similarity_score} = \alpha \cdot \text{Semantic} + \beta \cdot \text{Lexical} + \gamma \cdot \text{Syntactic} \quad (10)$$

where $\alpha + \beta + \gamma = 1$, and equal weights are used in this study. The similar sample top-ranked are selected from the demonstration pool and used as part of the ICL (Figure 3.), guiding LLMs toward more accurate and context-aligned code generation.

D. SMALL MODEL-AUGMENTED PROMPT

After constructing a custom CoT and selecting the ICL, the TIPS framework enters its final phase: assembling a prompt template that integrates the predictions of the small model y_i , confidence score c_i , and CoT reasoning chain to guide the LLMs in code generation tasks. This prompt template comprises two core components:

```
1 {
2   {
3     "code": "void* add(void* arg) {\n for (int i = 0; i < 1000; i++) {\n pthread_mutex_lock(&lock);\n count++;\n pthread_mutex_unlock(&lock);\n }\n return NULL;\n}",
4     "Small model Prediction": "critical_section" (Confidence: 88.58%),
5     "True label": "critical_section"
6   },
7   {
8     "code": "void run_no_lock(void* arg) {\n for (int i = 0; i < 1000; i++) {\n count++;\n }\n}",
9     "Small model Prediction": "unrelated" (Confidence: 91.38%),
10    "True label": "unrelated"
11  },
12  {
13    "code": "void* modify_balance(void* arg) {\n pthread_mutex_lock(&mutex);\n balance += 100;\n pthread_mutex_unlock(&mutex);\n return NULL;\n}",
14    "Small model Prediction": "critical_section" (Confidence: 84.29%),
15    "True label": "critical_section"
16  }
17 }
```

FIGURE 3. Code-level example of CP prediction results with confidence scores.

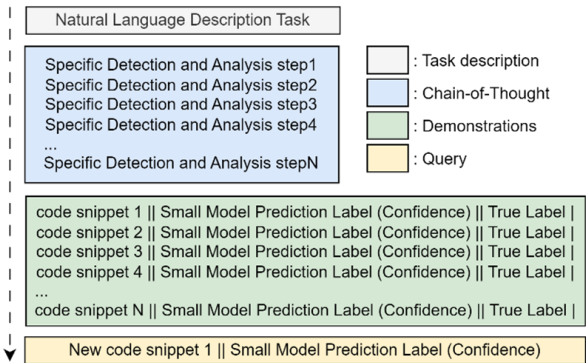


FIGURE 4. The final prompt format combining reasoning steps, demonstration examples, and query for LLM-based code analysis.

- **CoT:** Facilitate stepwise reasoning, improve LLM’s interpretability, and preventing illogical output from direct end-to-end generation.
- **Prediction with Confidence Score:** The small model output \hat{y}_i and the confidence score c_i from f_{SM} serve as prior signals. The LLMs adopts \hat{y}_i when c_i exceeds a threshold; otherwise, it relies on internal reasoning.

The complete prompt template enables the LLMs to handle ambiguous or conflicting code segments by contextually choosing between trusting the small model’s output or relying on its internal reasoning, as illustrated in Figure 4. This adaptive mechanism promotes correction under uncertainty. The TIPS framework not only improves accuracy and robustness in code generation but also improves interpretability, offering a reliable, flexible solution for automated strategy identification and parallel programming assistance.

IV. EXPERIMENTAL DESIGN

A. DATASETS

This study focuses on a common synchronization error in multithreaded programming: data races. These typically arise when developers fail to properly apply mutexes or other synchronization mechanisms, allowing multiple threads to concurrently access and modify shared resources, which may lead to race conditions and unpredictable behavior. We collect original Pthread source files (50–250 lines each) from

open-source GitHub repositories¹. All collected versions include correct mutex-based synchronization, ensuring thread safety in critical sections. To enable systematic evaluation, we curated two datasets based on distribution characteristics. Balanced dataset and an Imbalanced dataset.

- **Balanced dataset:** Contains equal numbers of correct and faulty samples. Faulty samples are created by injecting a single synchronization flaw (e.g. omission of `pthread_mutex_lock()` into correct code. This design minimizes distribution bias and isolates the impact of prompting strategies on classification and generation performance.
- **Imbalanced dataset:** Simulate real-world imbalance by including samples with 2–4 distinct data race defects (e.g. missing locks, incorrect unlocks, inadequate coverage). This dataset tests the generalization and robustness of the TIPS framework under diverse and imbalanced conditions.

B. BASELINE MODEL

To evaluate the performance of the TIPS framework in multithreaded code generation tasks, we selected reproducible baseline models including the Support Vector Machine (SVM) [72], AdaBoost [73], and Bagging [74]. These models are widely adopted in related tasks and datasets, offering a solid foundation for comparative analysis. Since our framework involves both error classification and code generation, the classification component focuses on the identification of synchronization errors, formulated as a supervised learning problem. Employing traditional classifiers allows for a fair comparison with LLM-based approaches, thereby isolating the added value brought by the TIPS framework.

SVM is a well-established supervised classification algorithm that finds an optimal hyperplane to maximize the margin between classes. It is especially effective in high-dimensional spaces and supports kernel methods such as linear, polynomial, and radial basis function (RBF) to handle non-linear decision boundaries. Due to its strong generalization capabilities, SVM has demonstrated robust performance in defect detection and software metrics prediction tasks.

AdaBoost is an ensemble learning algorithm that iteratively combines multiple weak learners, typically decision trees, while dynamically adjusting sample weights based on prior classification errors. The final predictions are determined through a weighted voting mechanism. AdaBoost is particularly effective in handling noisy or complex feature distributions and has been successfully applied in domains such as code intent detection and software defect prediction.

Bagging is a parallel ensemble technique that creates multiple training subsets via replacement with sampling. Each subset is used to train an independent classifier, and the final decisions are made by majority vote. This method effectively

¹The dataset is available at <https://github.com/MayAlsofyani/Pthread-Benchmark>

reduces model variance and improves stability, making it well suited for mitigating overfitting in classification tasks.

C. EVALUATION METRICS

To quantify the classification performance of the TIPS framework, we employ three standard metrics. Precision, recall, and F1-score. These are derived from the confusion matrix comprising True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN), and are also calculated for baseline models to ensure a fair and comparable evaluation.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (11)$$

This measures the proportion of positive predictions that are correct, reflecting the model's ability to avoid false alarms.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (12)$$

This indicates the coverage of the model of actual positive instances.

$$F1 \text{ Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (13)$$

The F1-Score is the harmonic mean of Precision and Recall and is particularly suitable for imbalanced class distributions.

D. RESEARCH QUESTIONS

To comprehensively evaluate the TIPS framework in the multithreaded code classification and generation, we designed a series of comparative experiments that focus on two key dimensions: classification performance and generation quality. These experiments aim to assess the effectiveness of the framework, quantify the contribution of its core components, and evaluate its robustness under varying data distributions.

For classification, we adopt SVM, AdaBoost, and Bagging as baseline models, representing traditional machine learning classifiers. These serve as a reference for comparing the integration of prompt engineering and LLM within TIPS. Given the diverse and imbalanced nature of real-world software bugs, we propose the following research questions.

- **RQ1:** Can the proposed TIPS Framework, using prompt engineering and LLM integration, outperform conventional classifiers in Pthread code classification and generation?
- **RQ2:** Does the TIPS Framework maintain stable reasoning and generation performance under imbalanced distributions and high code diversity?

To further explore the contribution of individual modules within TIPS, we conducted ablation studies by removing or replacing each core component to observe its effect on classification performance.

- **Small Model:** Generates initial class predictions during the demonstration phase, providing auxiliary information for the LLM.

- **CoT:** Guides the LLM through a logical reasoning process to systematically analyze code structure.
- **Demonstrations:** Provide examples most similar to the target sample, facilitating accurate inference in analogous contexts.

Accordingly, We investigate the following research question:

- **RQ3:** What individual contributions and effects do these core components offer in classification and generation tasks?

In the code generation phase, this study assesses whether the TIPS framework design significantly improves the accuracy of the LLM for Pthread code generation tasks. GPT-3.5 serves as the generation model, and we compare two prompting strategies:

- A TIPS-derived prompt template incorporating auxiliary reasoning information
- A Basic Prompt containing only the task description and input code, without additional context

This leads to the following research question:

- **RQ4:** Can the TIPS Framework, through its prompt engineering, significantly improve LLM accuracy in Pthread code generation?

V. RESULT

To assess whether the TIPS Framework outperforms conventional classifiers in classification tasks, we compared its performance against SVM, AdaBoost, and Bagging. Table 1 summarizes model performance across multiple Pthread task categories. The TIPS framework achieved *Precision* = 0.97, *Recall* = 0.90, and *F1-Score* = 0.94. significantly surpassing SVM(*Precision* = 0.80, *Recall* = 0.63, *F1-Score* = 0.78). Notably, for the categories Mutex Management and Condition Variables, TIPS achieved *F1-Scores* of 0.95 in both, indicating substantially improved recognition of critical synchronization strategies.

To evaluate the robustness of the TIPS framework under imbalanced data distributions, the experimental results are summarized in Table 2. On the Imbalanced Dataset, TIPS achieved a performance of *Precision* = 0.82, *Recall* = 0.42, and *F1-Score* = 0.57, outperforming the SVM(*Precision* = 0.42, 0.22, *F1-Score* = 0.28) and Bagging(*Precision* = 0.85, 0.41, *F1-Score* = 0.55).

To quantify the contributions of the core components in the TIPS framework, we conducted ablation experiments by removing the Small Model, CoT, and Demonstrations modules one at a time. Table 3 reports performance under each configuration.

- Without the Small Model (w/o Small Model), the framework's performance drops to *Precision* = 0.75, *Recall* = 0.50, and *F1-Score* = 0.58, significantly below the full-framework averages(*Precision* = 0.86, *Recall* = 0.62, *F1-Score* = 0.61).
- Removing the CoT module (w/o CoT) yields an overall performance similar to w/o Small Model (*Precision* =

TABLE 1. Performance comparison of models on pthread code generation tasks (Balanced Dataset).

| Task | SVM | | | Adaboost | | | Bagging | | | TIPS Framework | | |
|------------------------------|------|------|------|----------|------|------|---------|------|------|----------------|------|------|
| | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 |
| 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 | 0.80 | 0.63 | 0.78 | 0.89 | 0.82 | 0.87 | 0.82 | 0.88 | 0.89 | 0.97 | 0.90 | 0.94 |

Note. Pre = Precision; Re = Recall; F1 = F1 Score.

TABLE 2. Robustness evaluation of small models and the TIPS framework under imbalanced pthread task distribution.

| Task | SVM | | | Adaboost | | | Bagging | | | TIPS Framework | | |
|------------------------------|------|------|------|----------|------|------|---------|------|------|----------------|------|------|
| | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 |
| Thread Lifecycle | 0.53 | 0.05 | 0.17 | 0.88 | 0.05 | 0.20 | 0.88 | 0.15 | 0.33 | 0.77 | 0.12 | 0.32 |
| Mutex Management | 0.42 | 0.45 | 0.55 | 0.73 | 0.42 | 0.52 | 0.80 | 0.52 | 0.63 | 0.94 | 0.64 | 0.76 |
| Condition Variables | 0.00 | 0.00 | 0.00 | 0.45 | 0.54 | 0.49 | 0.70 | 0.59 | 0.64 | 0.85 | 0.45 | 0.60 |
| Thread Pool Creation | 0.48 | 0.02 | 0.05 | 0.91 | 0.52 | 0.67 | 1.00 | 0.26 | 0.51 | 0.71 | 0.28 | 0.44 |
| Scheduling Policy Assignment | 0.69 | 0.59 | 0.63 | 0.77 | 0.59 | 0.66 | 0.85 | 0.51 | 0.64 | 0.83 | 0.63 | 0.72 |
| Average | 0.42 | 0.22 | 0.28 | 0.75 | 0.42 | 0.49 | 0.85 | 0.41 | 0.55 | 0.82 | 0.42 | 0.57 |

Note. Pre = Precision; Re = Recall; F1 = F1 Score.

TABLE 3. Effectiveness of the TIPS framework under different component configurations.

| Task | w/o Small model | | | w/o CoT | | | w/o Demonstration | | | TIPS Framework | | |
|------------------------------|-----------------|------|------|---------|------|------|-------------------|------|------|----------------|------|------|
| | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 | Pre | Re | F1 |
| Thread Lifecycle | 0.40 | 0.50 | 0.44 | 0.25 | 0.25 | 0.25 | 0.29 | 0.50 | 0.36 | 0.20 | 0.50 | 0.29 |
| Mutex Management | 0.84 | 0.64 | 0.73 | 0.82 | 0.72 | 0.77 | 1.00 | 0.72 | 0.84 | 0.76 | 0.76 | 0.73 |
| Condition Variables | 0.74 | 0.58 | 0.65 | 0.70 | 0.58 | 0.64 | 0.68 | 0.62 | 0.65 | 0.82 | 0.75 | 0.78 |
| Thread Pool Creation | 1.00 | 0.22 | 0.37 | 1.00 | 0.29 | 0.45 | 0.92 | 0.28 | 0.43 | 0.93 | 0.29 | 0.45 |
| Scheduling Policy Assignment | 0.79 | 0.56 | 0.71 | 0.76 | 0.66 | 0.73 | 0.84 | 0.60 | 0.70 | 0.76 | 0.81 | 0.79 |
| Average | 0.75 | 0.50 | 0.58 | 0.71 | 0.50 | 0.59 | 0.75 | 0.54 | 0.60 | 0.86 | 0.62 | 0.61 |

Note. Pre = Precision; Re = Recall; F1 = F1 Score.

0.71, Recall = 0.50, F1-Score = 0.59). However, Recall for Thread Lifecycle and Thread Pool Creation drops to 0.25 and 0.29, indicating that CoT improves minority class detection by decomposing complex logic.

- Without demonstrations (w/o Demonstration), the F1-Score is 0.60, just below the full-framework value of 0.61. In particular, in the Mutex Management task, Precision increases from 0.70 to 1.00, suggesting that relevant examples can guide LLM toward the correct implementation of the synchronization strategy in specific contexts.

Figure 5 illustrates the performance of code generation. When employing a basic prompt, GPT-3.5 generates basic thread-safe structures, but omits essential syn-

chronization mechanisms such as `mutex_lock()` and `pthread_mutex_unlock()`. In contrast, the prompt template explicitly instructs the model to include these calls, ensuring the correct protection of shared resources.

VI. DISCUSSION

The TIPS framework significantly outperforms traditional classifiers in multithreaded code classification, particularly in Mutex Management and Condition Variables tasks. By combining CoT with small-model assistance, the TIPS framework reduces misclassification rates, aligning with findings from Ahmed and Devanbu [31] and Nashid et al. [30]. While their prompt retrieval strategy improves error correction, it suffers from low recall in complex, synchronization-heavy

scenarios. In contrast, the TIPS framework employs a small model to preselect candidate labels and simulates stepwise human reasoning via CoT, resulting in enhanced accuracy and stability across multiple defect categories. Moreover, unlike conventional classifiers, the TIPS framework provides not only label predictions but also an interpretable reasoning chain, addressing the lack of transparency in traditional approaches, as noted by Fan et al. [27].

Although performance declined across all models under the imbalanced dataset, the TIPS framework still outperformed most traditional classifiers, demonstrating its ability to mitigate degradation caused by data sparsity. Previous studies have similarly reported that both LLMs and traditional classifiers struggle with imbalanced or long-tail distributions. For example, Mišić and Dodović [39] found that GitHub Copilot exhibited unstable behavior when handling low-frequency synchronization errors, which required the support of external static analysis tools. Nashid et al. [30] further showed that recall rates for rare error types can drop by as much as 40% when using standard prompts, leading to increased omission rates in real-world applications. In contrast, the TIPS Framework utilizes a small model to pre-filter error types and enhances LLM reasoning via CoT and targeted demonstrations, providing contextual reinforcement. This integrated prompting strategy reduces the uncertainty of classification, resulting in improved precision and greater stability under skewed data distributions.

We further investigated the individual contributions of each TIPS component to multithreaded code classification, addressing a research gap in prompt engineering studies noted by Brown et al. [13]. Removing the Small Model caused substantial performance degradation, underscoring its role in guiding LLM inference and reducing misclassification. Excluding CoT led to a similar overall decline, with a marked drop in recall for Thread Lifecycle and Thread Pool Creation tasks, emphasizing CoT’s importance in rare class detection and multistep reasoning. Removing demonstrations slightly reduced the F1-score, although precision improved in Mutex Management tasks, suggesting that in-context examples assist with generating accurate synchronization strategies.

Finally, we evaluated the effectiveness of the TIPS framework in guiding code generation for multithreaded synchro-

nization tasks. Although prior work such as Ahmed and Devanbu [31] and Nashid et al. [30] demonstrated that example-based prompting enhances contextual coherence in general code generation, they lacked targeted evaluations for synchronization control. Mišić and Dodović [39] further noted that LLMs often omit critical section protection or misuse synchronization primitives in concurrent code. The TIPS Framework addresses this gap by integrating CoT reasoning with retrieved demonstrations, enabling LLMs to apply synchronization primitives correctly and avoid common protection omissions. This finding aligns with the logical CoT approach proposed by Zhao et al. [69], reinforcing the idea that structured stepwise reasoning improves safety and correctness in code generation.

VII. LIMITATIONS

This study presents several limitations. First, the dataset is primarily constructed from the Pthread-based open source project on GitHub. Although manually annotated and enhanced with injected synchronization errors, the dataset may not reflect broader language features and diverse architectural paradigms, such as Rust’s `async/await` mechanism or OpenCL’s multicore scheduling. Therefore, the generalizability of the TIPS framework across heterogeneous environments and programming languages requires further validation.

Second, although the TIPS framework improves inference controllability and efficiency by integrating small-model predictions and prompt design, its construction is heavily based on expert knowledge and manual effort. In particular, crafting CoT reasoning templates and selecting semantically similar demonstrations require significant domain expertise, which may hinder generalization across different tasks or domains and increase deployment cost.

Third, the overall effectiveness of the TIPS framework is influenced by the reasoning capabilities of the downstream LLM. When paired with a high-capacity model, comparable or superior performance can still be achieved without auxiliary commands. However, for models with limited reasoning ability, structured reasoning templates, semantic demonstration retrieval, and small-model filtering contribute significantly to improving inference accuracy and consistency.

VIII. CONCLUSION

Empirical evaluations demonstrate that the TIPS framework significantly outperforms conventional machine learning classifiers in standard classification benchmarks. In addition, it maintains robust generalization even under real-world challenges, such as imbalanced data distributions, mixed error types, and complex program logic. Ablation studies confirm the essential contributions of each core component: the multimodule design provides complementary capabilities that no single component can substitute.

In particular, utilizing a small model as a front-end classifier reduces the cognitive burden on the LLMs by directing attention to semantically salient regions, thereby facilitating

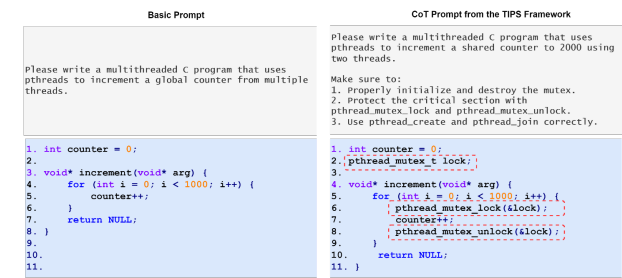


FIGURE 5. Comparison of Code Generation Results Using Basic Prompt vs. Prompt template from TIPS Framework.

prompt templates that are logically coherent and semantically aligned. These results empirically validate structured prompt design strategies and underscore the critical role of prompt flow engineering, not merely model scale or training data, in achieving enhanced reasoning performance.

In terms of practical contribution, the TIPS framework is tailored for code classification and generation in resource-constrained environments. By integrating a lightweight small model, in-context learning, and CoT reasoning, it strengthens semantic alignment and reasoning capability while reducing dependency on high-compute infrastructure. Its modular architecture supports low-cost integration with standard LLM APIs, enabling practical deployment in educational environments, software development tools, and semantic assistance systems. This provides a viable solution for improving the controllability and applicability of LLMs in real-world scenarios.

In terms of theoretical contribution, this work contributes to the theoretical advancement of prompt engineering by addressing two core aspects: structured prompt design and contextual adaptation. In contrast to conventional studies that focus on isolated strategies such as zero-shot, few-shot, or CoT, the proposed TIPS framework employs a multimodule pipeline integrating semantic similarity retrieval, small-model-based classification prescreening, reasoning chain construction, and confidence calibration. By explicitly modeling the interaction between LLMs capabilities and prompt dependency, this framework closes a gap in existing research regarding the mutual influence of model capacity and prompting strategies. Hence, it expands the theoretical landscape of prompt-based language model application design.

The TIPS framework reconceptualizes the capabilities of LLMs as not solely dependent on parameter scale, but rather enhanced through structured, semantic, and reasoning-informed prompt design. This framework presents a sustainable and flexible paradigm for the deployment of the language model. It signals a shift away from cloud-centric, opaque LLMs usage toward lightweight, locally deployable solutions suitable for education, development, and everyday applications. The overarching objective is to democratize intelligent technologies by facilitating their broader accessibility in real-world programming, instructional, and learning environments.

REFERENCES

- [1] L. Chen, A. Bhattacharjee, N. Ahmed, N. Hasabnis, G. Oren, V. Vo, and A. Jannesari, "Ompgpt: A generative pre-trained transformer model for openmp," in *European Conference on Parallel Processing*. Springer, 2024, pp. 121–134, https://doi.org/10.1007/978-3-031-69577-3_9.
- [2] M. A. Haque, "LLms: A game-changer for software engineers?" *Benchmark Transactions on Benchmarks, Standards and Evaluations*, p. 100204, 2025, <https://doi.org/10.1016/j.tbench.2025.100204>.
- [3] L. Cao and C. Dede, "Navigating a world of generative ai: Suggestions for educators," *The next level lab at harvard graduate school of education*, vol. 5, no. 2, 2023.
- [4] J. Steiss, T. Tate, S. Graham, J. Cruz, M. Hebert, J. Wang, Y. Moon, W. Tseng, M. Warschauer, and C. B. Olson, "Comparing the quality of human and chatgpt feedback of students' writing," *Learning and Instruction*, vol. 91, p. 101894, 2024.
- [5] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "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*, 2024, pp. 111–118, <https://doi.org/10.1145/3643795.3648379>.
- [6] C. S. Xia, Y. Wei, and L. Zhang, "Practical program repair in the era of large pre-trained language models," *arXiv preprint arXiv:2210.14179*, 2022, <https://doi.org/10.48550/arXiv.2210.14179>.
- [7] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "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*, 2022, pp. 39–51, <https://doi.org/10.1145/3533767.3534390>.
- [8] J. Cao, M. Li, M. Wen, and S.-c. Cheung, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *Automated Software Engineering*, vol. 32, no. 1, pp. 1–29, 2025, <https://doi.org/10.1007/s10515-025-00492-x>.
- [9] J. L. Espejel, M. S. Y. Alassan, E. M. Chouham, W. Dahhane, and E. H. Ettifouri, "A comprehensive review of state-of-the-art methods for java code generation from natural language text," *Natural Language Processing Journal*, vol. 3, p. 100013, 2023, <https://doi.org/10.1016/j.nlp.2023.100013>.
- [10] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for ai-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, 2023, <https://doi.org/10.3390/e25060888>.
- [11] X. Wu, R. Duan, and J. Ni, "Unveiling security, privacy, and ethical concerns of chatgpt," *Journal of information and intelligence*, vol. 2, no. 2, pp. 102–115, 2024, <https://doi.org/10.1016/j.jiixd.2023.10.007>.
- [12] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
- [13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [14] L. Guo, Y. Wang, E. Shi, W. Zhong, H. Zhang, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "When to stop? towards efficient code generation in llms with excess token prevention," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1073–1085, <https://doi.org/10.1145/3650212.3680343>.
- [15] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024, <https://doi.org/10.48550/arXiv.2401.04088>.
- [16] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv e-prints*, pp. arXiv–2407, 2024, <https://doi.org/10.48550/arXiv.2407.21783>.
- [17] J. Lin, J. Tang, H. Tang, S. Yang, G. Xiao, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *GetMobile: Mobile Computing and Communications*, vol. 28, no. 4, pp. 12–17, 2025, <https://doi.org/10.1145/3714983.3714987>.
- [18] E. Frantar, S. Ashkboos, T. Hoeffer, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022, <https://doi.org/10.48550/arXiv.2210.17323>.
- [19] P. Khosravi, "Querymate: A custom llm powered by llamacpp," 2024.
- [20] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021, <https://doi.org/10.48550/arXiv.2107.03374>.
- [21] M. Alier, F. J. G. Peñalvo, and J. D. Camba, "Generative artificial intelligence in education: From deceptive to disruptive," *International Journal of interactive multimedia and artificial intelligence*, vol. 8, no. 5, pp. 5–14, 2024, <https://doi.org/10.9781/ijimai.2024.02.011>.
- [22] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. T. Lee, "Textbooks are all you need ii: phi-1.5 technical report," *arXiv preprint arXiv:2309.05463*, 2023, <https://doi.org/10.48550/arXiv.2309.05463>.
- [23] C. Wu, B. Ma, Z. Zhang, N. Deng, Y. He, and Y. Xue, "Evaluating zero-shot multilingual aspect-based sentiment analysis with large language models," *International Journal of Machine Learning and Cybernetics*, pp. 1–23, 2025, <https://doi.org/10.1007/s13042-025-02711-z>.

- [24] M. Liu, F. Wu, B. Li, Z. Lu, Y. Yu, and X. Li, "Envisioning class entity reasoning by large language models for few-shot learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 18, 2025, pp. 18 906–18 914, <https://doi.org/10.1609/aaai.v39i18.34081>.
- [25] H. Yang, Q. Zhao, and L. Li, "Chain-of-thought in large language models: Decoding, projection, and activation," *arXiv preprint arXiv:2412.03944*, 2024, <https://doi.org/10.48550/arXiv.2412.03944>.
- [26] S. R. Hiraou, "Optimising hard prompts with few-shot meta-prompting," *arXiv preprint arXiv:2407.18920*, 2024, <https://doi.org/10.48550/arXiv.2407.18920>.
- [27] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53, <https://doi.org/10.1109/ICSE-FoSE59343.2023.00008>.
- [28] Y. Yang, Y. Zhu, S. Chen, and P. Jian, "Api comparison knowledge extraction via prompt-tuned language model," *Journal of Computer Languages*, vol. 75, p. 101200, 2023, <https://doi.org/10.1016/j.cola.2023.101200>.
- [29] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu, "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*, 2022, pp. 1–13, <https://doi.org/10.1145/3551349.3559612>.
- [30] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462, <https://doi.org/10.1109/ICSE48619.2023.00205>.
- [31] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, 2022, pp. 1–5, <https://doi.org/10.1145/3551349.3559555>.
- [32] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with llvm compiler: Compile-time instrumentation for threadsanitizer," in *International Conference on Runtime Verification*. Springer, 2011, pp. 110–114, https://doi.org/10.1007/978-3-642-29860-8_9.
- [33] D. Liew, T. Cogumbreiro, and J. Lange, "Sound and partially-complete static analysis of data-races in gpu programs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2434–2461, 2024, <https://doi.org/10.1145/3689797>.
- [34] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 258–269, <https://doi.org/10.1145/512529.512560>.
- [35] S. Malakar, T. B. Haider, and R. Shahriar, "Racefixer—an automated data race fixer," *arXiv preprint arXiv:2401.04221*, 2024, <https://doi.org/10.48550/arXiv.2401.04221>.
- [36] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. De Supinski, "Lm4hpc: Towards effective language model application in high-performance computing," in *International Workshop on OpenMP*. Springer, 2023, pp. 18–33, https://doi.org/10.1007/978-3-031-40744-4_2.
- [37] T. Kadosh, N. Schneider, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Advising openmp parallelization via a graph-based approach with transformers," in *International Workshop on OpenMP*. Springer, 2023, pp. 3–17, https://doi.org/10.1007/978-3-031-40744-4_1.
- [38] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. Cerpa, and W. Du, "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*, 2023, pp. 951–960, <https://doi.org/10.1145/3624062.3624172>.
- [39] M. Mišić and M. Dodović, "An assessment of large language models for openmp-based code parallelization: a user perspective," *Journal of Big Data*, vol. 11, no. 1, p. 161, 2024, <https://doi.org/10.1186/s40537-024-01019-z>.
- [40] M. Alsafyani and L. Wang, "Detecting data races in openmp with deep learning and large language models," in *Workshop Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 96–103, <https://doi.org/10.1145/3677333.3678160>.
- [41] D. Deanda, Y. P. Masupalli, J. Yang, Y. Lee, Z. Cao, and G. Liang, "Benchmarking robustness of contrastive learning models for medical image-report retrieval," *arXiv preprint arXiv:2501.09134*, 2025, <https://doi.org/10.48550/arXiv.2501.09134>.
- [42] Y. Zhang, G. Liang, T. Salem, and N. Jacobs, "Defense-pointnet: Protecting pointnet against adversarial attacks," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 5654–5660, <https://doi.org/10.1109/BigData47090.2019.9006307>.
- [43] X. He, W. Huang, and C. Lv, "Trustworthy autonomous driving via defense-aware robust reinforcement learning against worst-case observational perturbations," *Transportation Research Part C: Emerging Technologies*, vol. 163, p. 104632, 2024, <https://doi.org/10.1016/j.trc.2024.104632>.
- [44] D. Deanda, I. Alsmadi, J. Guerrero, and G. Liang, "Defending mutation-based adversarial text perturbation: a black-box approach," *Cluster Computing*, vol. 28, no. 3, p. 196, 2025, <https://doi.org/10.1007/s10586-024-04916-3>.
- [45] L. Solovyeva, S. Weidmann, and F. Castor, "Ai-powered, but power-hungry? energy efficiency of llm-generated code," in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 2025, pp. 49–60, <https://doi.org/10.1109/Forge66646.2025.00012>.
- [46] V. Bolón-Canedo, L. Morán-Fernández, B. Cancela, and A. Alonso-Betanzos, "A review of green artificial intelligence: Towards a more sustainable future," *Neurocomputing*, vol. 599, p. 128096, 2024, <https://doi.org/10.1016/j.neucom.2024.128096>.
- [47] T. Coignon, C. Quinton, and R. Rouvoy, "A performance study of llm-generated code on leetcode," in *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, 2024, pp. 79–89, <https://doi.org/10.1145/3661167.3661221>.
- [48] R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong, "How efficient is llm-generated code? a rigorous & high-standard benchmark," *arXiv preprint arXiv:2406.06647*, 2024, <https://doi.org/10.48550/arXiv.2406.06647>.
- [49] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "Effibench: Benchmarking the efficiency of automatically generated code," *Advances in Neural Information Processing Systems*, vol. 37, pp. 11 506–11 544, 2024.
- [50] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao *et al.*, "What's wrong with your code generated by large language models? an extensive study," *arXiv preprint arXiv:2407.06153*, 2024, <https://doi.org/10.48550/arXiv.2407.06153>.
- [51] W. Hou and Z. Ji, "Comparing large language models and human programmers for generating programming code," *Advanced Science*, vol. 12, no. 8, p. 2412279, 2025, <https://doi.org/10.1002/adv.202412279>.
- [52] C. Niu, T. Zhang, C. Li, B. Luo, and V. Ng, "On evaluating the efficiency of source code generated by llms," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 2024, pp. 103–107, <https://doi.org/10.1145/3650105.3652295>.
- [53] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng, "Mercury: A code efficiency benchmark for code large language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 16 601–16 622, 2024.
- [54] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025, <https://doi.org/10.1145/3703155>.
- [55] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-efficient fine-tuning for large models: A comprehensive survey," *arXiv preprint arXiv:2403.14608*, 2024, <https://doi.org/10.48550/arXiv.2403.14608>.
- [56] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022, <https://doi.org/10.48550/arXiv.2204.05999>.
- [57] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "StarCoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023, <https://doi.org/10.48550/arXiv.2305.06161>.
- [58] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023, <https://doi.org/10.48550/arXiv.2308.12950>.
- [59] C. Curto, D. Giordano, D. G. Indelicato, and V. Patatu, "Can a llama be a watchdog? exploring llama 3 and code llama for static application security testing," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2024, pp. 395–400, <https://doi.org/10.1109/CSR61664.2024.10679444>.
- [60] Z. Yu, Y. Zhao, A. Cohan, and X.-P. Zhang, "Humaneval pro and mbpp pro: Evaluating large language models on self-

invoking code generation,” *arXiv preprint arXiv:2412.21199*, 2024, <https://doi.org/10.48550/arXiv.2412.21199>.

[61] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi *et al.*, “Textbooks are all you need,” *arXiv preprint arXiv:2306.11644*, 2023, <https://doi.org/10.48550/arXiv.2306.11644>.

[62] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi *et al.*, “Phi-2: The surprising power of small language models,” *Microsoft Research Blog*, vol. 1, no. 3, p. 3, 2023.

[63] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, “Stanford alpaca: An instruction-following llama model,” 2023.

[64] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, “Gqa: Training generalized multi-query transformer models from multi-head checkpoints,” *arXiv preprint arXiv:2305.13245*, 2023, <https://doi.org/10.48550/arXiv.2305.13245>.

[65] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” *arXiv preprint arXiv:2004.05150*, 2020, <https://doi.org/10.48550/arXiv.2004.05150>.

[66] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023, <https://doi.org/10.48550/arXiv.2307.09288>.

[67] L. Tunstall, E. Beeching, N. Lambert, N. Rajani, K. Rasul, Y. Belkada, S. Huang, L. Von Werra, C. Fourrier, N. Habib *et al.*, “Zephyr: Direct distillation of lm alignment,” *arXiv preprint arXiv:2310.16944*, 2023, <https://doi.org/10.48550/arXiv.2310.16944>.

[68] S. Hu, Y. Tu, X. Han, C. He, G. Cui, X. Long, Z. Zheng, Y. Fang, Y. Huang, W. Zhao *et al.*, “Minicpm: Unveiling the potential of small language models with scalable training strategies,” *arXiv preprint arXiv:2404.06395*, 2024.

[69] X. Zhao, M. Li, W. Lu, C. Weber, J. H. Lee, K. Chu, and S. Wermter, “Enhancing zero-shot chain-of-thought reasoning in large language models through logic,” *arXiv preprint arXiv:2309.13339*, 2023, <https://doi.org/10.48550/arXiv.2309.13339>.

[70] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, “Self-consistency improves chain of thought reasoning in language models,” *arXiv preprint arXiv:2203.11171*, 2022, <https://doi.org/10.48550/arXiv.2203.11171>.

[71] J. Von Oswald, E. Niklasson, E. Randazzo, J. Sacramento, A. Mordvintsev, A. Zhmoginov, and M. Vladymyrov, “Transformers learn in-context by gradient descent,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 35 151–35 174.

[72] T. Joachims *et al.*, “Transductive inference for text classification using support vector machines,” in *Icml*, vol. 99, 1999, pp. 200–209.

[73] Y. Freund, R. E. Schapire *et al.*, “Experiments with a new boosting algorithm,” in *icml*, vol. 96. Bari, Italy, 1996, pp. 148–156.

[74] L. Breiman, “Bagging predictors,” *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996, <https://doi.org/10.1007/BF00058655>.



SHIH-YEH CHEN received his Ph.D. degree from the Department of Engineering Science, National Cheng Kung University, Tainan, Taiwan, in 2016. He is currently an Associate Professor in the same department. His research interests include digital twins, human-computer interaction, and learning support systems.

...



WEI-CHENG CHEN is a master’s degree student in the Department of Engineering Science, National Cheng Kung University, Tainan, Taiwan. His research interests include Artificial Intelligence of Things and Human-Computer Interaction.