Tuning LLM-based Code Optimization via Meta-Prompting: An Industrial Perspective

Jingzhi Gong[¶†], Rafail Giavrimis[‡†], Paul Brookes[†], Vardan Voskanyan[†], Fan Wu[†], Mari Ashiga[§†], Matthew Truscott[†], Mike Basios[†], Leslie Kanthan[†], Jie Xu[¶], and Zheng Wang[¶*]
[¶]University of Leeds, Leeds, UK
[†]TurinTech AI, London, UK
[‡]University of Surrey, Surrey, UK
[§]University of West London, London, UK
Emails: {j.gong, j.xu, z.wang5}@leeds.ac.uk, mari.ashiga@uwl.ac.uk,
{rafail, paul, vardan, fan, matthew.truscott, mike, leslie}@turintech.ai

***Abstract*—There is a growing interest in leveraging multiple large language models (LLMs) for automated code optimization. However, industrial platforms deploying multiple LLMs face a critical challenge: prompts optimized for one LLM often fail with others, requiring expensive model-specific prompt engineering. This cross-model prompt engineering bottleneck severely limits the practical deployment of multi-LLM systems in production environments. We introduce Meta-Prompted Code Optimization (MPCO), a framework that automatically generates high-quality, task-specific prompts across diverse LLMs while maintaining industrial efficiency requirements. MPCO leverages meta-prompting to dynamically synthesize context-aware optimization prompts by integrating project metadata, task requirements, and LLM-specific contexts. It is an essential part of the ARTEMIS code optimization platform for automated validation and scaling.**

Our comprehensive evaluation on five real-world codebases with 366 hours of runtime benchmarking demonstrates MPCO's effectiveness: it achieves overall performance improvements up to 19.06% with the best statistical rank across all systems compared to baseline methods. Analysis shows that 96% of the top-performing optimizations stem from meaningful edits. Through systematic ablation studies and meta-prompter sensitivity analysis, we identify that comprehensive context integration is essential for effective meta-prompting and that major LLMs can serve effectively as meta-prompters, providing actionable insights for industrial practitioners.

***Index Terms*—Meta-Prompting, Code Optimization, Prompt Engineering, Performance Optimization, LLM4Code, AI4SE**

I. INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities in many aspects across the software development lifecycle, including code completion [1], natural-language documentation [2], test-case generation [3], automated debugging [4], and more [5]. Beyond these capabilities, recent research studies have demonstrated that LLMs can systematically analyze code patterns to uncover optimization opportunities and produce revised implementations, ranging from memory-management enhancements and algorithmic refinements to automated parallelization and vectorization—thereby enabling high-performance, resource-efficient software [6]–[8].

*Corresponding author: Zheng Wang (z.wang5@leeds.ac.uk).

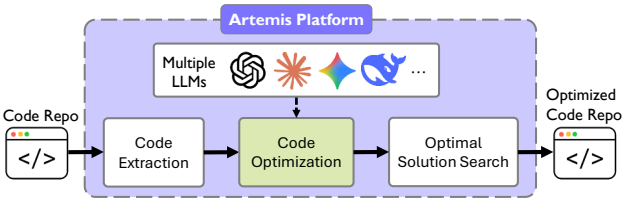

Fig. 1: ARTEMIS' multi-LLM code optimization workflow.

The success of LLM-based code optimization has led to the development of industrial solutions that integrate these capabilities into production workflows. These platforms provide enterprise-grade infrastructure for deploying LLM-driven code optimization at scale, serving diverse client needs from high-frequency trading systems requiring microsecond latencies to scientific computing applications demanding maximum throughput [9]–[13]. Among others, TurinTech AI's ARTEMIS platform [14] represents a leading AI-driven software performance engineering system, which deploys multiple LLMs at scale to search for optimal optimizations of critical code snippets with real-time validation, as illustrated in Figure 1. It now processes tens of thousands of requests daily and offers automated performance-engineering solutions that reduce development costs and accelerate time-to-market for performance-critical applications [15].

However, in its production deployment, ARTEMIS exposes a fundamental obstacle in multi-LLM code optimization, which we term the "*cross-model prompt engineering challenge*" - a prompt that produces excellent optimizations with one LLM may yield poor results with others [16], [17]. Consequently, platforms need to develop and maintain model-specific prompts for each LLM-task combination; furthermore, industrial platforms often face strict efficiency constraints that limit the feasibility of advanced prompt optimization methods that require extensive trial-and-error or computation [17]. This raises the critical question for this work:

How can we automatically generate high-quality,

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00295

task-specific prompts across diverse code optimization LLMs while maintaining industrial efficiency requirements?

Unfortunately, existing solutions cannot adequately address these requirements. Conventional prompting studies like Chain-of-Thought (CoT) focus primarily on general-purpose tasks and single-model scenarios, lacking the specialized contextual awareness and cross-model adaptability required for industrial code optimization [17]. Evolutionary prompt optimization methods require pre-defined evaluation functions for each project and multi-round processing, causing scalability problems [18], [19]. Advanced prompt tuning agents, while more flexible and powerful, demand extensive computing resources, impractical for production environments requiring rapid processing [20], [21].

To address this, we introduce Meta-Prompted Code Optimization (MPCO), an end-to-end framework that automates prompt engineering while maintaining industrial requirements for flexibility and efficiency across multiple LLMs. In summary, our key contributions are:

- We present a novel prompt engineering framework that feeds a meta-prompting LLM rich contextual information to efficiently generate model-adaptive prompts, thereby addressing the cross-model prompting challenge that limits industrial platforms.
- We demonstrate MPCO's effectiveness through comprehensive evaluation across multiple LLMs on the ARTEMIS platform using five real-world codebases and three major LLMs. Each optimization is verified through 10 repeated benchmarking, with total validation time exceeding 336 hours.
- Through detailed ablation studies and meta-prompter sensitivity analysis, we identify that comprehensive context integration is essential for effective meta-prompting, and that all three major LLMs can serve effectively as meta-prompters, providing actionable recommendations for industrial deployment.

II. PRELIMINARIES

A. Industrial Multi-LLM Code Optimization

Code optimization aims to improve software performance (e.g., making program run fast or reducing the binary size) via algorithmic and low-level code transformations. It has traditionally relied on handcrafted algorithmic tweaks, compiler-driven pre-defined transformations (e.g., loop unrolling, inlining, and dead-code elimination), and machine learning–based autotuning systems [22]. Yet, these methods often struggle to capture domain-specific performance patterns, generalize across diverse codebases, and incur high search overheads [22], [23]. Recent advances in LLMs enable automated code optimization systems that learn from vast code corpora and leverage both a static knowledge base and environment interaction to generate adaptive, context-aware optimization suggestions, without expert intervention or limitation on pre-defined operations [6]. Today, industrial platforms integrate these LLM-driven optimizations into production pipelines, offering low-latency inference and high-throughput processing [9], [14].

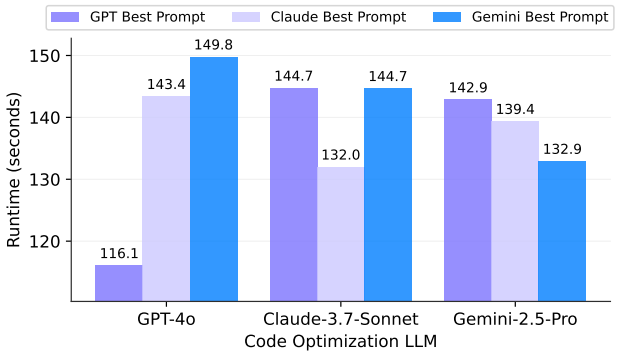

Fig. 2: Cross-model optimization results on LLAMA.CPP.

However, single-LLM approaches cannot meet these diverse industrial requirements effectively, as different models exhibit complementary strengths that are needed for comprehensive optimization coverage [11]. Consequently, industrial platforms like ARTEMIS deploy multiple LLMs simultaneously to leverage their diverse capabilities [11], [15]. Specifically, Figure 1 illustrates the ARTEMIS workflow, where the codebase is first analyzed to identify optimization targets, then sent to a diverse set of LLMs for improvement suggestions, and finally filtered through an intelligent search to find the optimal solution.

B. Cross-Model Prompt Engineering Bottleneck

The effectiveness of LLM-generated optimizations depends critically on prompt quality [16], [17], yet different models exhibit varying sensitivities to prompt structure, context, and optimization strategies. As a motivating example, consider optimizing a hot code region of the LLAMA.CPP inference framework [24]. Figure 2 shows that optimizing the same code snippet with three LLMs using different optimal prompt templates reveals significant performance variations. Especially, GPT-4o achieves its best performance (116.1s) with its own prompt template, but degrades by 23.5% (143.4s) when using Claude's template and 29.0% (149.8s) with Gemini's template. These inconsistencies often stem from model-specific factors - differences in training corpora and architectures, variations in tokenization and context-window handling, and divergent decoding defaults and instruction-following behaviors [17], [25], [26].

To achieve reliable performance across multiple LLMs, platforms often have to optimize and maintain distinct prompts for every LLM-task-project pairing. The resulting multiplicative complexity renders traditional prompting workflows infeasible in production environments where rapid, large-scale, generalizable optimizations are required.

This challenge is uniquely severe in code optimization compared to generic natural language processing (NLP) tasks. Unlike text generation, where outputs can be stylistically

TABLE I: Comparison of prompting approaches. ✓= Full support, ∼= Partial support, ✗= No support

Approach	Context-Awareness	End-to-End Automation	Deployment Simplicity	Low Overhead	Cross-Metric Adaptability	Cross-Model Adaptability	Cross-Project Adaptability
Manual Prompting	∼	✗	✓	✗	✗	✗	✗
Baseline Prompting	✗	✓	✓	✓	✗	✗	✗
Evolutionary Prompting	✗	✓	∼	✗	∼	∼	∼
Agentic Systems	✓	✓	✗	✗	✓	✓	✓
MPCO (Our Approach)	✓	✓	✓	✓	✓	✓	✓

varied, optimized code has strict, non-negotiable correctness requirements: it must compile, pass all functional tests, and be free of runtime side effects like memory leaks. The determinism and reproducibility of optimizations are paramount. A slight, seemingly innocuous change in a prompt can lead to a syntactically correct but functionally flawed or slower implementation, making the cross-model challenge a critical barrier to reliable industrial deployment.

C. Existing Prompt Optimization Approaches and Limitations

Several approaches have been proposed to automate prompt engineering, but each exhibits critical limitations when applied to multi-LLM industrial code optimization scenarios, as summarized in Table I.

Baseline prompting techniques include the most common prompt designs such as direct instructions (simple and ad-hoc task specifications), Chain-of-Thought reasoning (step-by-step problem decomposition) [27], few-shot learning (task demonstration through examples) [28], and contextual prompting (fixed template with rich task-related contexts) [29]. While efficient and easy to deploy, these methods rely on static, manually-crafted templates that cannot adapt seamlessly across different optimization metrics, models, or project types.

Evolutionary prompting methods encompass both automatic prompt optimization approaches such as EvoPrompt (genetic algorithm-based prompt evolution) [18] and APE (automatic prompt engineering through iterative refinement) [26], as well as gradient-based prompt tuning techniques (continuous optimization of prompt embeddings) [16], [19], [30]. While these methods achieve substantial gains for well-defined tasks, they require multiple rounds of LLM interaction, leading to high computation overhead [18], which could be infeasible for platforms processing tens of thousands of daily requests [15]. Furthermore, they often rely on manual design of evaluation pipelines for each task/project, limiting their scalability in production environments.

Agentic prompting systems employ autonomous AI agents that iteratively refine prompts through environmental interaction, feedback incorporation, and self-reflection mechanisms [31], [32]. These agent-based approaches demonstrate context-awareness and cross-domain adaptability by dynamically adjusting prompts based on task requirements and model responses. However, they usually incur higher deployment complexity and overhead due to extensive environment interaction cycles, action planning, agent state updates, and memory I/O [33], making them unsuitable for large-scale industrial scenarios requiring rapid responses.

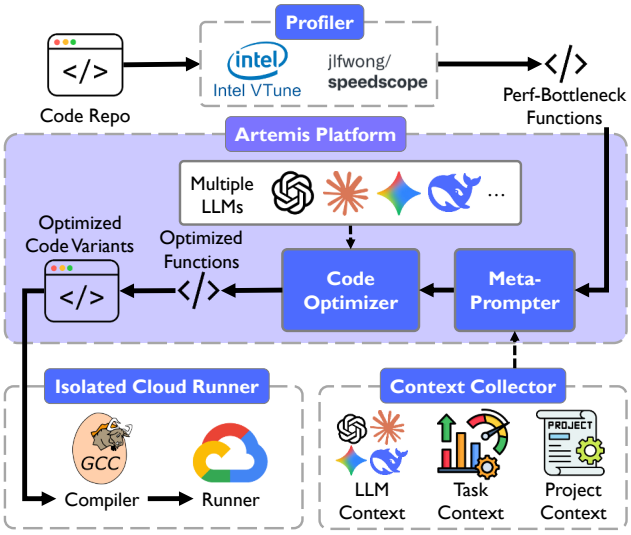

Fig. 3: The end-to-end workflow of MPCO.

Existing meta-prompting approaches [34]–[38] employ higher-order prompts to autonomously generate task-specific instructions, and have shown promise in domains such as text classification [37], dialog response generation [35], [38], and mathematical reasoning [34]. However, these approaches may either generate generic prompts without context-awareness, lack end-to-end deployment efficiency, or have not been validated on real-world industrial environments.

III. THE MPCO FRAMEWORK

To address these limitations, we propose Meta-Prompted Code Optimization (MPCO), a meta-prompting framework specifically designed for multi-LLM industrial code optimization. As summarized in Table I, MPCO differs from existing approaches by systematically incorporating task-specific contexts, providing efficient and scalable deployment, and integrating with automated performance validation requiring minimal human intervention.

The architecture of MPCO (Figure 3) consists of four core stages that ensure complete automation from profiling to validated optimization results, as delineated below.

A. Stage 1: Profiling and Bottleneck Identification

The framework begins by systematically identifying performance bottlenecks within real-world software projects. In this stage, users specify software projects requiring optimization,

and the framework applies industry-standard profilers - Intel VTune Profiler [39] for C++ and Speedscope [40] for Python - to locate performance-critical code snippets. These profiling tools use instrumentation to trace function calls and resource usage at runtime, and visualize hotspots via flame graphs or detailed reports. We opt for VTune and Speedscope in this study because VTune provides low-overhead, cycle-accurate profiling of C++ code [41], while Speedscope offers rapid and efficient exploration for dynamic Python workloads [42].

We then rank the identified performance bottlenecks by their impact on runtime and forward the top-10 code snippets to the next stage, enabling our framework to prioritize efficiency-critical code rather than arbitrary selections. Formally, this profiling process can be defined as:

$$\overline{\boldsymbol{\mathcal{B}}} = \texttt{TopK}(\texttt{Profile}(\boldsymbol{\mathcal{R}}), k = 10), \tag{1}$$

where $\boldsymbol{\mathcal{R}}$ is the input code repository, `Profile` is the profile function, `TopK` selects the k functions with the highest performance impact, and $\overline{\boldsymbol{\mathcal{B}}} = \{\boldsymbol{\mathcal{B}}_1, \boldsymbol{\mathcal{B}}_2, \ldots, \boldsymbol{\mathcal{B}}_{10}\}$ is the set of top-10 bottleneck code snippets selected for optimization.

B. Stage 2: Context Collection and Meta-Prompt Generation

To overcome the cross-model prompt engineering bottleneck, MPCO introduces a novel meta-prompting mechanism that swiftly collects and unifies comprehensive relevant context for prompt synthesis. Unlike prior works that either use generic templates without context awareness [34], [38] or incur large overhead in context collection [18], [32], MPCO builds on Artemis' existing metadata store to enable rapid context gathering through three lightweight contexts:

1) **Project context** extracts project metadata (name, description, and languages) from Artemis's existing project database, ensuring the generated prompt maintains syntactic validity and project consistency.
2) **Task context** maintains optimization-specific requirements (target metrics, domain constraints, and optimization considerations) in a simple JSON format that updates dynamically as new performance goals emerge.
3) **LLM context** stores model-specific characteristics (target LLM, model strengths and limitations) in database tables that scale with new model deployments.

This Artemis-integrated context storage enables context collection in seconds, without the computational overhead of traditional and more advanced prompt optimization methods, and also enables easy maintenance and updates, allowing the framework to scale up seamlessly as the platform evolves with new model deployments and optimization requirements.

The meta-prompting LLM receives structured contexts through the MPCO template (Figure 4) and produces specialized prompts in a single inference call, eliminating the need for per-model manual tuning while maintaining industrial-grade efficiency. Formally, this process can be defined as a two-step process:

$$\boldsymbol{\mathcal{P}}_{m,t,p} = \texttt{GenPrompt}(\boldsymbol{\mathcal{T}}(\boldsymbol{\mathcal{C}}_m, \boldsymbol{\mathcal{C}}_t, \boldsymbol{\mathcal{C}}_p)), \tag{2}$$

```
You are an expert in code optimization. Please
    generate a prompt that will instruct the
    target LLM {target_llm} to optimize code for
    {objective}.
Consider the project context, task context, and
    adapt the prompt complexity and style based
    on the target LLM's capabilities.

## Project Context
Project Name: {project_name}
Project Description: {project_description}
Primary Languages: {project_languages}

## Task Context
- Description: {task_description}
- Considerations: {task_considerations}

## Target LLM Context
- Target Model: {target_llm}
- Considerations: {llm_considerations}
```

Fig. 4: The MPCO meta-prompting template.

where $\boldsymbol{\mathcal{C}}_m, \boldsymbol{\mathcal{C}}_t, \boldsymbol{\mathcal{C}}_p$ represent model, task, and project contexts respectively, $\boldsymbol{\mathcal{T}}$ is the meta-prompt template, `GenPrompt` is the meta-prompting function, and $\boldsymbol{\mathcal{P}}_{m,t,p}$ is the generated prompt for model m, task t, and project p.

C. Stage 3: Multi-LLM Code Optimization

In this stage, generated meta-prompts are deployed across multiple target LLMs simultaneously to leverage diverse optimization capabilities [11]. For each bottleneck function $\boldsymbol{\mathcal{B}}_i$ identified in Stage 1, each LLM receives its model-specific prompt along with the performance-critical code snippet, producing code optimizations that reflect both the model's inherent strengths and the contextual guidance provided by the meta-prompt.

For each optimized code snippet, the framework then generates a variant of the original code repository where only one specific code snippet is optimized, while preserving the complete project context including build configurations, dependency relationships, and test suites. This ensures that the optimized code can be validated under identical conditions to the original implementation, addressing a critical requirement for performance evaluation.

The optimization process is implemented within the Artemis platform, which orchestrates the multi-LLM deployment and manages the code variant generation workflow. Formally, the process can be defined as:

$$\boldsymbol{\mathcal{O}}_i = \texttt{LLM}(\boldsymbol{\mathcal{P}}_{m,t,p}, \boldsymbol{\mathcal{B}}_i), \tag{3}$$

$$\boldsymbol{\mathcal{V}}_i = \texttt{GenVariant}(\boldsymbol{\mathcal{R}}, \boldsymbol{\mathcal{O}}_i), \tag{4}$$

where $\boldsymbol{\mathcal{B}}_i$ is a bottleneck code snippet, $\boldsymbol{\mathcal{P}}_{m,t,p}$ is the generated prompt, `LLM` is the code optimization function, $\boldsymbol{\mathcal{R}}$ is the original code repository, `GenVariant` is the code variant generation function, $\boldsymbol{\mathcal{O}}_i$ and $\boldsymbol{\mathcal{V}}_i$ are the optimized code snippet and code repository variant for bottleneck $\boldsymbol{\mathcal{B}}_i$, respectively.

D. Stage 4: Performance Evaluation and Validation

Finally, the framework automatically validates the generated code variants through isolated cloud services provided by ARTEMIS. The validation process includes: compiling all code variants to ensure syntactic correctness, executing comprehensive unit test suites to verify functional equivalence, running performance benchmarks, collecting detailed runtime metrics, and statistical significance analysis.

This systematic validation ensures that only functionally correct and performance-validated optimizations are accepted, maintaining the reliability standards required for industrial deployment. Formally, this process can be defined as:

$$\boldsymbol{\mathcal{E}}_i = \texttt{Validate}(\boldsymbol{\mathcal{V}}_i), \tag{5}$$

where $\boldsymbol{\mathcal{V}}_i$ is a code variant from Stage 3, `Validate` is the function that performs compilation, unit testing, and benchmarking, and $\boldsymbol{\mathcal{E}}_i$ is the evaluated performance result.

IV. EXPERIMENT SETUP

A. Research Questions

To evaluate the effectiveness of MPCO, we conducted comprehensive experiments to answer the following research questions (RQs):

- **RQ1**: How does MPCO address the cross-model prompt engineering bottleneck compared to baseline prompting approaches?
- **RQ2**: What aspects of MPCO's context collection most significantly impact its effectiveness?
- **RQ3**: How sensitive is MPCO to the choice of meta-prompting LLM?

B. Subject Systems and Bottleneck Selection

We systematically selected five subject systems from open-source GitHub repositories: we identified candidate repositories through GitHub searches for PRs labeled "*runtime optimization*", "*speedup*", or "*runtime improvement*" [43], and filtered them by requiring built-in benchmarks, clear build systems, and target language support. Our study concentrated on C++ and Python projects, as these languages are dominant in open-source performance-critical systems and are widely supported by state-of-the-art LLMs and profiling tools [39], [40], [44]. For each project, we selected the top-10 most performance-critical code snippets based on the profiling data.

This selection strategy ensures that our evaluation covers the most relevant industrial scenarios where performance optimization is critical [43]. Table II summarizes the selected systems, including project name, description, language, and evaluation benchmark.

C. Performance Evaluation and Validation

Each generated code optimization was integrated into a new code variant, and each variant was submitted to Artemis for automated compilation and execution. Any LLM response that failed to adhere to the required format (containing only the synthesized prompt/optimized code without additional commentary) was excluded from analysis. Similarly, code optimizations that resulted in compilation errors or runtime failures were systematically removed from the dataset prior to statistical analysis. This filtering process ensures that our performance comparisons are based solely on functionally correct and executable code variants, providing a fair and meaningful evaluation of optimization effectiveness across different prompting approaches.

TABLE II: Summary of subject systems.

System	Description	Lang.	Benchmark
BitmapPlusPlus	Single-header BMP image library	C++	Built-in chess_board bench
Llama.cpp	Efficient LLM inference engine	C++	Built-in bench with TinyLlama
RPCS3	PlayStation 3 emulator	C++	Built-in rpcs3_benchmark
Faster-Whisper	Fast speech-to-text transcription	Python	Built-in transcription bench
Langflow	Visual flow-based LLM workflows	Python	Built-in benchmark_operations

To control measurement noise, we isolated the experiment environment and executed each benchmark 10 times, ensuring statistical significance and stable performance distributions. The LLMs used were GPT-4o (OpenAI) [45], Gemini 2.5 Pro (Google) [46], and Claude 3.7 Sonnet (Anthropic) [47], selected to represent the three major LLM providers and diverse architectural approaches. We used their default temperature settings to ensure reproducible results and avoid introducing additional variance from configuration tuning.

All experiments were conducted on a dedicated Google Cloud Platform server with 4 CPUs, 16GB RAM, running Ubuntu 25.04. The benchmarks were executed in Docker containers to ensure consistent execution environments and eliminate interference between different optimization runs. Each experiment was executed in an isolated environment without any other processes running to minimize measurement noise and ensure reproducible results.

D. Evaluation Metric and Statistical Analysis

To evaluate each code optimization, we adopt the most common metric in the LLM-based code optimization literature [6], i.e., the percentage runtime performance improvement (%PI), computed as:

$$\%PI \;=\; \frac{T_{orig} - T_{opt}}{T_{orig}} \;\times 100\%\,, \tag{6}$$

where T_{orig} is the mean runtime of the original code and T_{opt} is the mean runtime of the optimized code version.

We further conduct rigorous statistical analysis to rank prompt variants using Mann-Whitney U test [48] and Cohen's d effect size [49]. In particular, the Mann-Whitney U test—an assumption-free, non-parametric method suited to our independent, continuous %PI measurements—is applied to detect shifts in central tendency between optimization approaches, and Cohen's d is computed to standardize the observed mean differences and confirm practical relevance.

Our ranking strategy works as follows: (1) calculate the mean %PI for each approach, (2) sort approaches by mean %PI in descending order, and (3) assign ranks sequentially—for each approach, if it is statistically significantly different

TABLE III: The mean and standard deviation of %PI, denoted as Mean (SD), for MPCO and the basic prompting approach across three LLMs and five projects. For each case, green cells mean MPCO has the best mean %PI; or red cells otherwise. The one(s) with the best rank (r) from the statistical tests is in bold.

System	MPCO		CoT		Few-shot		Contextual	
	r	Mean (SD)	r	Mean (SD)	r	Mean (SD)	r	Mean (SD)
BITMAPPLUSPLUS	**1**	**19.06** (1.48)	**1**	**19.01** (1.53)	**1**	**19.29** (2.37)	2	18.49 (1.17)
LLAMA.CPP	**1**	**7.84** (10.52)	3	3.95 (1.60)	4	3.90 (2.00)	2	3.97 (2.01)
RPCS3	**1**	**4.44** (8.54)	2	2.39 (7.72)	3	0.77 (5.19)	**1**	**3.52** (6.46)
FASTER-WHISPER	**1**	**5.64** (3.58)	4	1.58 (5.78)	3	4.08 (2.91)	2	4.14 (1.66)
LANGFLOW	**1**	**9.01** (2.33)	2	2.72 (2.26)	4	1.63 (1.68)	3	2.34 (2.61)
Average r	**1.00**		2.40		3.00		2.00	

TABLE IV: The mean and standard deviation of %PI, denoted as Mean (SD), for MPCO and its ablated versions across three LLMs and five projects. For each case, green cells mean MPCO has the best mean %PI; or red cells otherwise. The one(s) with the best rank (r) from the statistical tests is in bold.

System	MPCO		MPCO NP		MPCO NT		MPCO NL	
	r	Mean (SD)	r	Mean (SD)	r	Mean (SD)	r	Mean (SD)
BITMAPPLUSPLUS	**1**	**19.06** (1.48)	**1**	**19.18** (2.22)	**1**	**19.14** (2.01)	**1**	**19.11** (1.95)
LLAMA.CPP	**1**	**7.84** (10.52)	4	1.85 (0.48)	2	2.10 (1.59)	3	1.99 (0.56)
RPCS3	**1**	**4.44** (8.54)	**1**	**4.30** (9.39)	2	2.45 (7.49)	**1**	**3.82** (10.31)
FASTER-WHISPER	**1**	**5.64** (3.58)	3	-0.37 (2.88)	2	-0.36 (6.48)	**1**	**5.73** (1.76)
LANGFLOW	**1**	**9.01** (2.33)	2	2.03 (3.21)	3	1.87 (2.82)	4	1.22 (1.02)
Average r	**1.00**		2.20		2.00		2.00	

from the previous approach ($p \leq 0.05$ or $|d| \geq 0.2$)[1], assign it the next rank; otherwise, assign it the same rank as the previous approach. By doing so, we group similar-performing methods together while distinguishing methods with meaningful performance differences.

V. EVALUATION

This section presents the results of our experiment, structured around the research questions defined in Section IV.

A. RQ1: MPCO *Effectiveness vs. Baseline Prompting*

1) Method: In industrial platforms, code optimizations are often expected to complete within seconds and at minimal computational cost. The advanced prompting techniques discussed in Section II, which often rely on extensive reasoning chains or agent coordination [18], [31], are thus beyond the scope of this study. Instead, we benchmarked MPCO against established baseline prompting methods most commonly used in LLM-based code optimization [6][2], including:

- **Chain-of-Thought (CoT):** Structured prompts that guide the LLM through step-by-step optimization [27].
- **Few-Shot Prompting:** Prompts incorporating optimization examples to demonstrate desired output patterns [28].
- **Contextual Prompting:** Prompts with the same context information as MPCO but without meta-prompting [29].

The evaluation procedures followed the methodology detailed in Section IV. Each method was applied to optimize all bottleneck code snippets across the five subject systems (50 in total) and across three target LLMs; for each optimization, we generated a code variant, measured its runtime through 10 benchmark executions (resulting in up to 1500 measurements per method), and conducted statistical tests (Mann-Whitney U test and Cohen's d effect size) to rank different approaches.

2) Results: Table III presents the results for **RQ1**. Clearly, MPCO achieves the best average rank (1.00) for all subject systems, significantly outperforming all baseline methods and demonstrating consistent gains across the three LLMs used for code optimization.

[1]The 0.05 significance level follows established norms in SE research [50], while the 0.2 threshold for effect size ensures that differences considered are not only statistically significant but also practically relevant [51].

[2]The detailed implementation of all baseline prompts is available here.

In particular, MPCO yielded the highest %PI in four out of five systems, with strong results on BITMAPPLUSPLUS (19.06%), LANGFLOW (9.01%), and LLAMA.CPP (7.84%). Notably, on BITMAPPLUSPLUS, both CoT and Few-shot prompting performed similarly to MPCO, and all three methods shared rank 1. This is likely due to the relatively standard and simple nature of the performance issues in BITMAPPLUSPLUS, which can be effectively addressed by the predefined patterns in few-shot learning prompts or by the reasoning capabilities introduced by CoT.

Interestingly, we can see that even when the same contexts were provided, contextual prompting did not achieve competitive results as MPCO. This highlights the strength of meta-prompting, which dynamically adjusts its guidance based on the specific task and the capabilities of the LLM. In contrast, contextual prompting may overwhelm the LLM with too much information, leading to degraded performance - similar to a teacher who uses the same teaching style for all students, regardless of their individual characteristics. In summary:

> ***RQ1:*** MPCO *consistently outperforms baseline prompting methods, achieving the top rank across all systems and the highest mean %PI in 4/5 systems, demonstrating that its context-aware meta-prompting strategy can effectively address the cross-model prompt engineering challenge.*

B. RQ2: Ablation Analysis of Contextual Components

1) Method: To assess the contribution of each contextual component in MPCO's template, we conducted a systematic ablation analysis by comparing the full template (Figure 4) against three reduced variants:

- **MPCO NP**: No project context (project name, description, and primary languages).
- **MPCO NT**: No task context (task description and optimization considerations).
- **MPCO NL**: No LLM context (model-specific characteristics and adaptation instructions).

The evaluation followed the same methodology as **RQ1**, applying each ablated version to all 50 code snippets across the five subject systems and three target LLMs.

2) Results: Table IV highlights the importance of using comprehensive contexts to MPCO's effectiveness, where the full MPCO template is consistently ranked the first across all systems and gets the highest %PI in three out of five, with performance gains up to 9.01%. Even in the two cases where the full MPCO template did not yield the highest %PI - FASTER-WHISPER and BITMAPPLUSPLUS - MPCO still achieved rank 1 based on statistical tests. These results demonstrate that removing any contextual component noticeably reduces MPCO's overall effectiveness, underscoring the importance of comprehensive context integration.

For FASTER-WHISPER, MPCO $_{\text{NL}}$ slightly outperformed the full template (5.73% vs. 5.64%), possibly because the optimization targets in this project are relatively well-known, making them less sensitive to model-specific instructions. In BITMAPPLUSPLUS, all variants, including the full template and ablated versions, shared the top rank, suggesting that optimization challenges in this system are relatively straightforward, and even limited contextual guidance is sufficient, which aligns with our findings from **RQ1** that performance issues in BITMAPPLUSPLUS appeared easier to diagnose.

Interestingly, the average ranks of the ablated variants were similar (2.20, 2.00, and 2.00), indicating that no single contextual component dominates in importance, but rather that their joint combination provides the most consistent benefit. Based on these observations, we have:

> ***RQ2:*** *The full* MPCO *template yields the highest %PI in 3/5 systems and best average rank (1.00), demonstrating that comprehensive context integration is essential for effective meta-prompting.*

C. RQ3: Sensitivity Analysis of Meta-Prompting LLM

1) Method: To answer this, we investigated MPCO's sensitivity to the choice of meta-prompting LLM by evaluating three different LLM configurations:

- **MPCO $_{4o}$**: Use GPT-4o as the meta-prompter.
- **MPCO $_{37}$**: Use Claude 3.7 Sonnet as the meta-prompter.
- **MPCO $_{25}$**: Use Gemini 2.5 Pro as the meta-prompter.

Unlike **RQ1**, where all methods shared the same meta-prompter (GPT-4o), this RQ evaluated three different meta-prompting LLMs, but all code optimizations were performed by the same optimizer (Claude 3.7 Sonnet) to ensure a controlled comparison.

2) Results: Table V presents the analysis regarding meta-prompting LLM choice. Notably, MPCO $_{4o}$ achieved the best average rank (1.00) across all systems and the highest %PI in three of them, with performance improvements up to 19.77%. This demonstrates that GPT-4o as a meta-prompter can generate highly effective optimization prompts.

Similarly, MPCO $_{25}$ maintained a competitive average rank of 1.20 (rank 1 in four out of five systems), with the best mean %PI in two systems, suggesting that Gemini 2.5 Pro may be better suited for certain optimization scenarios, likely due to its specialized pretraining and reasoning capabilities.

TABLE V: The mean and standard deviation of %PI, denoted as Mean (SD), for different meta-prompting LLMs across five projects. For each case, green cells mean the approach has the best mean %PI. The one(s) with the best rank (r) from the statistical tests is in bold.

System	MPCO $_{4o}$		MPCO $_{37}$		MPCO $_{25}$	
	r	Mean (SD)	r	Mean (SD)	r	Mean (SD)
BITMAPPLUSPLUS	**1**	**19.77** (2.01)	2	19.30 (1.39)	**1**	**19.50** (1.85)
LLAMA.CPP	**1**	**5.30** (0.33)	2	4.97 (0.99)	**1**	**5.38** (0.40)
RPCS3	**1**	**0.40** (1.11)	2	0.19 (0.26)	**1**	**0.63** (1.25)
FASTER-WHISPER	**1**	**3.17** (3.74)	3	0.20 (6.92)	2	2.06 (3.58)
LANGFLOW	**1**	**9.64** (3.32)	2	8.60 (1.67)	**1**	**9.31** (1.93)
Average r	**1.00**		2.20		1.20	

These findings have practical implications for industrial deployment: while GPT-4o shows a slight performance advantage, all three meta-prompting LLMs provide significant improvements over the original code. Therefore, organizations can confidently choose their meta-prompting LLM based on practical considerations, such as cost, availability, and platform compatibility, without significantly compromising MPCO's effectiveness. Consequently, we conclude:

> ***RQ3:*** MPCO $_{4o}$ *ranks best across all systems with with the highest %PI in 3/5 systems. However, while GPT-4o offers a slight advantage, all three models yield significant gains, confirming that* MPCO*'s benefits are not tied to a specific meta-prompter.*

VI. DISCUSSION

A. Practical Validity of MPCO Performance Improvements

Our evaluation across five real-world systems demonstrates that MPCO's performance improvements are both significant and robust. However, a critical concern in automated code optimization is whether performance improvements represent genuine algorithmic enhancements rather than just removing the original functions.

To understand the nature of MPCO's optimizations, we analyzed the optimizations with the top-10 code optimization for each subject system (50 analyses in total). Specifically, their distribution of optimization types is summarized in Figure 5.

Notably, Loop & Vectorization (26%), Algorithmic Improvements (20%), and Code Quality & Correctness improvements (20%) are the most targeted optimizations, whereas the small proportion (4%) of trivial optimizations, such as reducing the number of iterations and adjusting line breaks, indicates that MPCO effectively identifies common performance bottlenecks [6], [28] and applies appropriate optimization strategies.

B. Illustrative Examples of Generated Meta-Prompts

To illustrate MPCO's superiority over baseline methods, we examine concrete examples from LLAMA.CPP, where MPCO achieved 97.5% higher %PI over the best baseline (7.87 vs 3.97 in Table III).

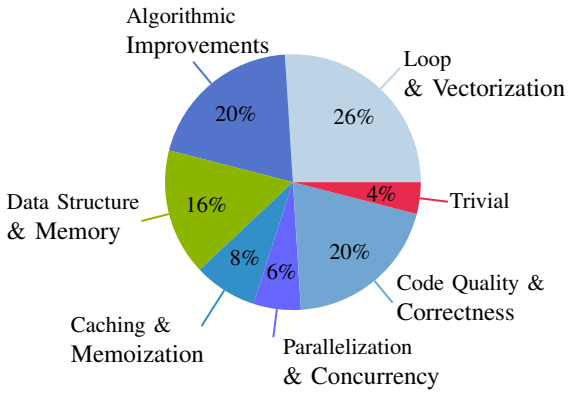

Fig. 5: Distribution of optimization types among the top 10% performing optimizations generated by MPCO.

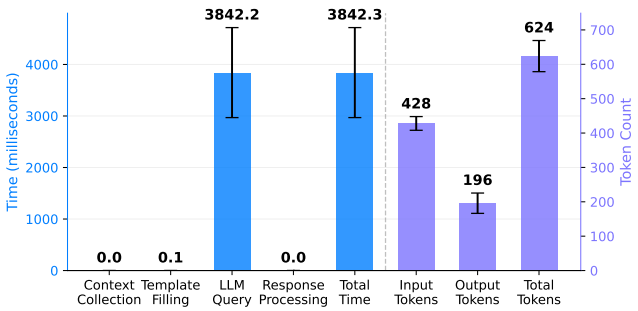

Fig. 6: The time and token overhead of MPCO.

Particularly, MPCO generates prompts with comprehensive optimization strategies, while adapting to each LLM's strengths (Appendix VIII). For example: GPT-4o (Figure 7) focuses on "complex interdependencies and comprehensive optimization"; Claude 3.7 Sonnet (Figure 8) promotes "systematic architectural thinking with maintainability considerations; and Gemini 2.5 Pro (Figure 9) emphasizes "complex reasoning to verify performance metrics".

In contrast, baseline methods provide generic guidance: chain-of-thought offers step-by-step frameworks without domain expertise (Figure 10); few-shot learning provides simple examples insufficient for C/C++ optimization (Figure 11); and contextual prompting uses all available contexts (project, task, LLM) uniformly across all cases (Figure 12).

Therefore, the key advantage of MPCO lies in its dual adaptation strategy: (1) **task-specific guidance** that addresses the unique characteristics of LLAMA.CPP (C/C++ codebase, performance-critical AI inference, memory-intensive operations), and (2) **LLM-specific instructions** tailored to each model's cognitive strengths and architectural capabilities. This explains why MPCO achieved top-ranked performance across all systems in **RQ1**, while baseline methods failed to consistently leverage each LLM's optimization capabilities or address task-specific performance bottlenecks.

C. Meta-Prompting Overhead Analysis

To validate MPCO's industrial efficiency claims, we conducted a comprehensive overhead analysis by measuring meta-prompting overhead across 50 constructs from all five subject systems, using GPT-4o as the meta-prompter. Figure 6 presents the detailed timing breakdown and token usage analysis.

The results demonstrate that MPCO introduces minimal computational overhead: averaging 3.8 seconds per meta-prompt generation with 624 tokens (428 input, 196 output). In particular, the LLM query dominates the timing, while context collection, template filling, and response processing contribute negligible overhead (<0.1ms each). At GPT-4o pricing ($2.5/1M input tokens, $10/1M output tokens), each meta-prompt generation costs approximately $0.003, making the approach highly practical for industrial deployment.

Importantly, MPCO provides substantial cost advantages over traditional prompt engineering approaches. Manual prompt crafting requires expert engineers to spend minutes to hours developing and tuning LLM-specific prompts for each optimization task, followed by ongoing maintenance as LLMs evolve. In contrast, MPCO automates this entire process at $0.003 per LLM-task pair, orders of magnitude lower cost than evolutionary prompt optimization or iterative agent-based methods that require multiple LLM calls. The single-call design ensures that once generated, meta-prompts can be reused indefinitely without additional overhead, reducing both the expertise requirement and maintenance burden while delivering superior optimization effectiveness across diverse tasks and LLMs.

D. Actionable Guidelines for Context Provision

Our ablation study in Section V-B offers clear insights into how context should be structured for effective meta-prompting in industrial environments. Particularly, the full MPCO template achieved the best average rank across all systems, while removing any contextual component noticeably reduces effectiveness. This demonstrates that MPCO performs best when supplied with comprehensive information, including target LLM capabilities, optimization objectives and considerations, and project metadata. Therefore, we have:

> ***Suggestion 1:*** *Maintain comprehensive context integration (project, task, and LLM context), as removing any contextual component might significantly degrade performance, with project context removal causing the highest degradation.*

In our implementation of MPCO, we developed a modular context-integration pipeline that queries the platform's existing APIs to retrieve contextual data from the database. By leveraging native services, we avoided duplicating functionality and maintained low-overhead data access. Furthermore, the contexts were stored using a lightweight, structured JSON schema that can be incrementally extended and updated as projects evolve, without requiring extensive re-engineering of the prompting infrastructure. As such:

Suggestion 2: *Use a modular, API-driven pipeline that integrates with existing platform services to enable rapid context-collection, and represent context in a structured schema to support flexible, incremental extension as the project evolves.*

E. Actionable Guidelines for Meta-Prompting LLM Selection

The comprehensive evaluation across multiple LLM configurations in Section V-C provides practical recommendations for organizations deploying MPCO in industrial environments. They show that all three major LLMs can serve effectively as meta-prompters, with GPT-4o achieving the best overall performance (average rank of 1.00) and Gemini 2.5 Pro also highly competitive (average rank of 1.20). Nevertheless, since the performance differences are relatively small, platforms can confidently base their meta-prompting LLM choice on pragmatic factors such as cost constraints, model availability, and integration requirements without compromising MPCO's effectiveness. In summary:

Suggestion 3: *Prioritize deployment considerations (cost, availability, integration) over minor performance differences when selecting meta-prompting LLMs, as all three major providers offer comparable effectiveness.*

Beyond the current one-step meta-prompting, platforms could also explore advanced meta-prompting techniques to enhance optimization effectiveness. For example, agent-based meta-prompting, where multiple specialized agents collaborate to dynamically collect contexts based on the task [31], and iterative meta-prompting, where prompts are refined through multiple rounds of prompt engineering [34]. While incurring additional computation and coordination overhead, these techniques can handle more complicated optimization scenarios that involve multiple interdependent code blocks or require dynamic and domain-specific contexts. Based on the above, we have:

Suggestion 4: *Consider advanced meta-prompting techniques (agent-based, iterative) as alternatives to one-step meta-prompting when platforms want to trade off computational resources for enhanced optimization performance.*

F. Limitations and Future Work

While MPCO demonstrates significant effectiveness in addressing the cross-model prompt engineering bottleneck, several limitations warrant discussion.

Language and domain: Our evaluation focuses on C++ and Python in specific domains (HPC, quantitative finance, data processing), and generalization to other languages and domains requires further validation. Future work could evaluate the effectiveness of meta-prompting across diverse programming languages and application domains to establish broader generalizability.

Optimization granularity: Currently, we only address function-level performance bottlenecks rather than file-level or system-level optimizations that require coordinated changes across multiple functions or entire systems. Future work could extend MPCO to handle multi-level optimization scenarios, developing context collection strategies that capture inter-function dependencies and system-wide performance characteristics.

Optimization target: This study targets runtime optimization and validation, without exploring other objectives like CPU usage, memory usage, and energy efficiency. Future work could easily expand MPCO's capabilities to handle different metric optimization scenarios, or even develop multi-objective meta-prompting strategies that can balance competing optimization goals.

Verification reliability: Our validation of functional correctness relies on existing unit test suites of target projects—if original test coverage is low, LLM-generated optimized code might introduce subtle bugs or edge case failures that go undetected. Future work should investigate integrating automatic test case generation, property-based testing, or symbolic checks to create more robust verification pipelines that are less dependent on existing test coverage.

VII. CONCLUSION

We have presented MPCO, a meta-prompting approach that automatically optimizes and selects effective prompts for code optimization across multiple LLMs. Over 366 hours of benchmarking, MPCO demonstrated significant performance gains on five real-world codebases while meeting industrial efficiency requirements. This makes it practical for automated code optimization platforms such as ARTEMIS to adopt multi-LLM optimization strategies without the burden of manual prompt engineering.

Based on our analysis, we recommend maintaining comprehensive context integration, automating context collection using existing infrastructure, selecting meta-prompting LLMs according to deployment constraints rather than marginal model differences, and applying advanced techniques only when higher performance justifies the additional computational cost. Our future work will extend MPCO to support multi-level and multi-objective optimization scenarios, as well as integrating automatic test generation for more robust verification pipelines.

ACKNOWLEDGMENT

This work was supported in part by an Innovative UK Knowledge Transfer Partnership (KTP) project between the University of Leeds and TurinTech AI, and the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreement EP/X037304/1.

REFERENCES

[1] R. A. Husein, H. Aburajouh, and C. Catal, "Large language models for code completion: A systematic literature review," *Computer Standards & Interfaces*, vol. 92, p. 103917, 2025.

[2] S. S. Dvivedi, V. Vijay, S. L. R. Pujari, S. Lodh, and D. Kumar, "A comparative analysis of large language models for code documentation generation," in *Proceedings of the 1st ACM international conference on AI-powered software*, 2024, pp. 65–73.

[3] S. Jorgensen, G. Nadizar, G. Pietropolli, L. Manzoni, E. Medvet, U.-M. O'Reilly, and E. Hemberg, "Large language model-based test case generation for gp agents," in *Proceedings of the genetic and evolutionary computation conference*, 2024, pp. 914–923.

[4] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," *Empirical Software Engineering*, vol. 30, no. 2, p. 45, 2025.

[5] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[6] J. Gong, V. Voskanyan, P. Brookes, F. Wu, W. Jie, J. Xu, R. Giavrimis, M. Basios, L. Kanthan, and Z. Wang, "Language models for code optimization: Survey, challenges and future directions," *arXiv:2501.01277*, 2025.

[7] S. Ishida, G. Corrado, G. Fedoseev, H. Yeo, L. Russell, J. Shotton, J. F. Henriques, and A. Hu, "Langprop: A code optimization framework using large language models applied to driving," 2024.

[8] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," *arXiv preprint arXiv:2407.02524*, 2024.

[9] Amazon Web Services, "AWS CodeGuru," https://aws.amazon.com/codeguru/, 2025, accessed: 2025-08-01.

[10] H. Lin, M. Maas, M. Roquemore, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa *et al.*, "Eco: An llm-driven efficient code optimizer for warehouse scale computers," *arXiv preprint arXiv:2503.15669*, 2025.

[11] R. Giavrimis, M. Basios, F. Wu, L. Kanthan, and R. Bauer, "Artemis ai: Multi-llm framework for code optimisation," in *2025 IEEE Conference on Artificial Intelligence (CAI)*. IEEE, 2025, pp. 1–6.

[12] U. Engineering, "Perfinsights: Detecting performance optimization opportunities in go code using generative ai," 2025, accessed: 2025-07-26. [Online]. Available: https://www.uber.com/blog/perfinsights/

[13] Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, and Y. Wu, "Aide: Ai-driven exploration in the space of code," *arXiv preprint arXiv:2502.13138*, 2025.

[14] TurinTech, "Artemis: Ai-powered code optimization platform," 2025. [Online]. Available: https://www.turintech.ai/artemis/

[15] BusinessWire, "Artemis launches at nvidia gtc to tackle ai-driven technical debt," March 2025, available at: link.

[16] A. Sabbatella, A. Ponti, I. Giordani, A. Candelieri, and F. Archetti, "Prompt optimization in large language models," *Mathematics*, vol. 12, no. 6, p. 929, 2024.

[17] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[18] Q. Chen, J. Tang, W. Liang, Y. Li, Y. Qiao, and X. Zhao, "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," *arXiv preprint arXiv:2309.08532*, 2023.

[19] R. Pryzant, D. Iter, J. Li, Y. T. Lee, C. Zhu, and M. Zeng, "Automatic prompt optimization with" gradient descent" and beam search," *arXiv preprint arXiv:2305.03495*, 2023.

[20] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *arXiv:2409.02977*, 2024.

[21] Z. Duan and J. Wang, "Exploration of llm multi-agent application implementation based on langgraph+ crewai," *arXiv preprint arXiv:2411.18241*, 2024.

[22] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, pp. 1879–1901, 2018.

[23] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[24] G. Gerganov and the community, "llama.cpp: Llm inference in c/c++," https://github.com/ggml-org/llama.cpp, 2023, version(s) as of access date. [Online]. Available: https://github.com/ggml-org/llama.cpp

[25] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM computing surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[26] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *The eleventh international conference on learning representations (ICLR)*, 2022.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[28] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," in *The Twelfth International Conference on Learning Representations, ICLR*. OpenReview.net, 2024.

[29] D. Huang, J. Dai, H. Weng, P. Wu, Y. Qing, H. Cui, Z. Guo, and J. Zhang, "Effilearner: Enhancing efficiency of generated code via self-optimization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 84 482–84 522, 2024.

[30] W. Chen, S. Koenig, and B. Dilkina, "Reprompt: Planning by automatic prompt engineering for large language models agents," *arXiv preprint arXiv:2406.11132*, 2024.

[31] X. Wang, C. Li, Z. Wang, F. Bai, H. Luo, J. Zhang, N. Jojic, E. P. Xing, and Z. Hu, "Promptagent: Strategic planning with language models enables expert-level prompt optimization," *arXiv preprint arXiv:2310.16427*, 2023.

[32] J. Zhang, Z. Wang, H. Zhu, J. Liu, Q. Lin, and E. Cambria, "Mars: A multi-agent framework incorporating socratic guidance for automated prompt optimization," *arXiv preprint arXiv:2503.16874*, 2025.

[33] H. Wang, J. Gong, H. Zhang, and Z. Wang, "Ai agentic programming: A survey of techniques, challenges, and opportunities," *arXiv preprint arXiv:2508.11126*, 2025.

[34] Y. Zhang, Y. Li, Z. Wang, J. Tan, G. Chen, J. Fan, Z. Hu, D. Liu, X. Zheng, J. Zhao *et al.*, "Meta prompting for ai systems," *arXiv preprint arXiv:2311.11482*, 2023.

[35] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 534–46 594, 2023.

[36] C. Gong, X. Li, J. Yu, Y. Cheng, J. Tan, and C. Yu, "Self-pro: A self-prompt and tuning framework for graph neural networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2024, pp. 197–215.

[37] Y. Hou, H. Dong, X. Wang, B. Li, and W. Che, "Metaprompting: Learning to learn better prompts," *arXiv preprint arXiv:2209.11486*, 2022.

[38] M. Suzgun and A. T. Kalai, "Meta-prompting: Enhancing language models with task-agnostic scaffolding," *arXiv preprint arXiv:2401.12954*, 2024.

[39] Intel Corporation, "Intel vtune profiler," https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html, 2021.

[40] C. J. Russell, "Speedscope: An interactive flame graph viewer," https://github.com/jlfwong/speedscope, 2020.

[41] PatSnap Eureka Blog, "Cpu profiling tools: Vtune vs perf vs gprof," https://eureka.patsnap.com/article/cpu-profiling-tools-vtune-vs-perf-vs-gprof, July 2025.

[42] S. Hanselman, "dotnet-trace for .net core tracing in perfview & speedscope: Chromium event trace profiling, flame graphs and more," Blog post, hanselman.com (2020-09-18), 2020, Link.

[43] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[44] TIOBE Software BV, "Tiobe programming community index," https://www.tiobe.com/tiobe-index/, July 2025.

[45] OpenAI, "Say hello to gpt-4o," https://openai.com/index/hello-gpt-4o/, May 2024.

[46] K. Kavukcuoglu, "Gemini 2.5: Our most intelligent ai model," https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/, Mar. 2025.

[47] Anthropic, "Claude 3.7 sonnet and claude code," https://www.anthropic.com/news/claude-3-7-sonnet, Feb. 2025.

[48] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.
[49] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Hillsdale, NJ: Routledge, 1988.
[50] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 1–10.
[51] G. M. Sullivan and R. Feinn, "Using effect size—or why the P value is not enough," *Journal of Graduate Medical Education*, vol. 4, no. 3, pp. 279–282, 2012. [Online]. Available: https://doi.org/10.4300/JGME-D-12-00156.1

VIII. Appendix: Prompt Examples

```
Optimize the C code in the llama.cpp project to
    enhance runtime performance by focusing on
    the following key areas:

1. **Algorithmic Complexity**: Analyze the code to
    identify and reduce the Big O complexity
    where possible. Consider alternative
    algorithms that offer better performance.
2. **Data Structure Efficiency**: Evaluate the
    current data structures for efficiency in
    terms of access patterns and memory usage.
    Replace or modify them to improve performance
    .
3. **Loop Optimizations**: Inspect loops for
    unnecessary iterations and optimize them.
    Consider loop unrolling and other techniques
    to reduce overhead.
4. **Memory Access Patterns**: Improve memory
    access patterns to enhance cache utilization.
    Minimize cache misses by optimizing data
    locality.
5. **I/O Operations**: Reduce the overhead of I/O
    operations and system calls. Batch operations
    where possible to minimize latency.
6. **Parallel Processing**: Identify opportunities
    for parallel processing. Use multithreading
    or SIMD instructions to leverage multiple
    cores and vectorization.
7. **Redundant Computations**: Eliminate redundant
    computations by caching results or reusing
    previous calculations where applicable.

For larger models like gpt-4-o, ensure to
    internally verify assumptions and metrics
    based on the task description, and consider
    complex interdependencies within the code for
    comprehensive optimization.
```

Fig. 7: MPCO generated prompt for runtime performance optimization in Llama.cpp targeting GPT-4o.

```
Optimize the C code in the llama.cpp project to
    enhance runtime performance through
    systematic analysis and targeted improvements
    :

1. **Algorithmic Complexity**: Examine the
    fundamental algorithms and identify
    opportunities to reduce computational
    complexity. Prioritize algorithmic changes
    that offer the greatest performance gains.
2. **Data Structure Optimization**: Assess current
    data structures for access efficiency and
    memory footprint. Consider restructuring data
    layouts to improve cache performance and
    reduce memory overhead.
3. **Loop and Control Flow**: Analyze loop
    structures for optimization opportunities
    including unrolling, vectorization, and
    elimination of unnecessary branching. Focus
    on hot paths and frequently executed code
    sections.
4. **Memory Management**: Optimize memory access
    patterns to maximize cache efficiency.
    Consider memory prefetching, data alignment,
    and reducing memory fragmentation.
5. **System-Level Optimizations**: Evaluate I/O
    operations, system calls, and resource
    utilization. Implement batching and buffering
    strategies to minimize system overhead.
6. **Concurrency and Parallelization**: Identify
    parallelizable sections and implement
    appropriate threading or SIMD optimizations.
    Balance parallelization overhead with
    performance gains.
7. **Code Efficiency**: Eliminate redundant
    computations, optimize critical path
    execution, and implement result caching where
    beneficial.

For claude-3-7-sonnet, approach optimization with
    systematic architectural thinking,
    considering both micro-optimizations and
    broader structural improvements. Provide
    clear rationale for each optimization
    decision and consider the maintainability
    implications of proposed changes.
```

Fig. 8: MPCO generated prompt for runtime performance optimization in Llama.cpp targeting Claude 3.7 Sonnet.

```
Optimize the C code in the llama.cpp project to
    enhance runtime performance by addressing the
     following aspects:

1. **Algorithmic Complexity**: Analyze and reduce
    the Big O complexity where possible. Identify
     and refactor inefficient algorithms.
2. **Data Structures**: Evaluate the efficiency of
     current data structures. Optimize access
    patterns and consider alternatives that offer
     better performance.
3. **Loop Optimizations**: Minimize unnecessary
    iterations. Unroll loops where beneficial and
     eliminate redundant calculations within
    loops.
4. **Memory Access Patterns**: Improve cache
    utilization by optimizing memory access
    patterns. Ensure data locality and minimize
    cache misses.
5. **I/O Operations**: Streamline I/O operations
    and reduce the frequency of system calls.
    Buffer data effectively to minimize latency.
6. **Parallel Processing**: Identify sections of
    the code that can be parallelized. Utilize
    multi-threading or SIMD instructions to
    leverage multi-core processors.
7. **Redundant Computations**: Detect and
    eliminate redundant computations. Cache
    results of expensive operations when possible
     to avoid repeated calculations.

For the gemini-v25-pro model, apply complex
    reasoning to verify assumptions about
    performance metrics and ensure optimizations
    align with the project's goals.
```

Fig. 9: MPCO generated prompt for runtime performance optimization in Llama.cpp targeting Gemini 2.5 Pro.

```
Let's optimize the following code step by step:

Please follow these reasoning steps:
1. First, analyze the current code to identify
    performance bottlenecks
2. Consider different optimization strategies (
    algorithmic, data structure, loop
    optimization, etc.)
3. Evaluate the trade-offs of each approach
4. Select the best optimization strategy
5. Implement the optimized version

Think through each step, then provide only the
    final optimized code.
```

Fig. 10: Chain-of-Thought (CoT) prompting method used as baseline.

```
Here are examples of code optimization:

Example 1 - Loop optimization:
Original: for i in range(len(arr)): if arr[i] >
    threshold: result.append(arr[i])
Optimized: result = [x for x in arr if x >
    threshold]

Example 2 - Algorithm optimization:
Original: for i in range(n): for j in range(n): if
     matrix[i][j] > 0: count += 1
Optimized: count = np.sum(matrix > 0)

Example 3 - Data structure optimization:
Original: items = []; for x in data: items.append(
    x); return sorted(items)
Optimized: return sorted(data)

Now optimize the code for better runtime
    performance, then provide only the final
    optimized code.
```

Fig. 11: Few-shot learning prompting method used as baseline.

```
You are an expert in code optimization. Please
    optimize the provided code for {objective}.
    Consider the project context, task context,
    and adapt your optimization approach
    accordingly.
## Project Context
Project Name: {project_name}
Project Description: {project_description}
Primary Languages: {project_languages}

## Task Context
- Description: {task_description}
- Considerations: {task_considerations}

## Target LLM Context
- Target Model: {target_llm}
- Considerations: {llm_considerations}
```

Fig. 12: Contextual prompting template used as baseline method, which uses all available contexts but without meta-prompting adaptation (the context placeholders will be filled in real prompts).