Prompt Engineering for Competitive Programming Problems

Team Members: Prakriti Adhikari, Fiza Ashraf, Aymen Noor

Emails: adhikap5@miamioh.edu, ashraff@miamioh.edu, noora@miamioh.edu

1. Introduction & Problem Definition

Problem Definition and Research Question

Large Language Models (LLMs) excel at many code generation tasks but frequently fail in highly reasoning-intensive domains, such as competitive programming. These problems demand complex, multi-step logical deduction, robust algorithm selection, and error-free code synthesis—areas where general LLMs often fall short.

The primary task of this project is to investigate and systematically compare the effect of various prompt engineering strategies on LLM performance when solving complex algorithmic problems. The central research question is: Which prompt design techniques (Zero-Shot, Few-Shot, Chain-of-Thought variants) most significantly enhance an LLM's logical reasoning and correctness in synthesizing complex, runnable algorithmic code?

Motivation and Real-World Relevance

The ability of an AI system to reliably solve complex, reasoning-based coding tasks is critical for advanced developer tooling, automated code review, and educational platforms. By identifying optimal prompting strategies, this research directly contributes to improving the robustness and reliability of AI in high-stakes technical domains, pushing their capability from simple syntax generation toward being true problem-solving partners.

Connections to Course Topics

This project is fundamentally aligned with the course topics of Prompting, Reasoning, and Evaluation. The entire study is centered on systematically testing prompt engineering variants like standard zero-shot, few-shot, and Chain-of-Thought (CoT) to improve LLM reasoning. The quantitative evaluation framework, which measures code correctness against hidden unit tests, provides an objective assessment of the effectiveness of these strategies.

2. Dataset

Dataset Sources and Structure

The project utilized the APPS (Automated Programming Problem Set) dataset from UC Berkeley, a standard benchmark for evaluating LLM code generation. The dataset was selected for its rigorous structure, which enables objective, test-case-based evaluation.

The entire evaluation was conducted exclusively on a subset of the APPS test dataset. Each problem provides four necessary components:

- question.txt: The detailed problem description used as the LLM input.
- input_output.json: Provides the official Ground Truth Test Cases for objective evaluation.
- Difficulty/Filtering: All selected problems are uniformly tagged with the difficulty Interview.

B. Preprocessing and Filtering Steps

Due to significant computational and API constraints, the final operational dataset was restricted to the first 100 problems sampled sequentially from the APPS test dataset. This non-random sequential sampling method ensures the experiment is fully reproducible. Running the full suite of prompt strategies across the primary model required over 10 hours of computation, reinforcing the necessity of this size constraint.

C. Ethical Considerations

Area	**Implication and Mitigation**
Model Bias	LLMs are trained on public codebases, which can perpetuate algorithmic biases based on common or non-optimal coding practices found in the training data. The use of a standardized competitive programming dataset partially mitigates this by focusing on objectively correct, efficient algorithms.
Data Privacy	The APPS dataset consists of abstract algorithmic problems and public solutions, meaning the study does not handle any sensitive personal data or private code. Privacy risk is therefore negligible.
Licensing	The APPS dataset is publicly available for research. The study uses standard open-source Python libraries (e.g., NumPy) and commercial APIs (Gemini) under their respective licenses. All generated code is subject to the project's own research licensing.

3. Methods / System Design

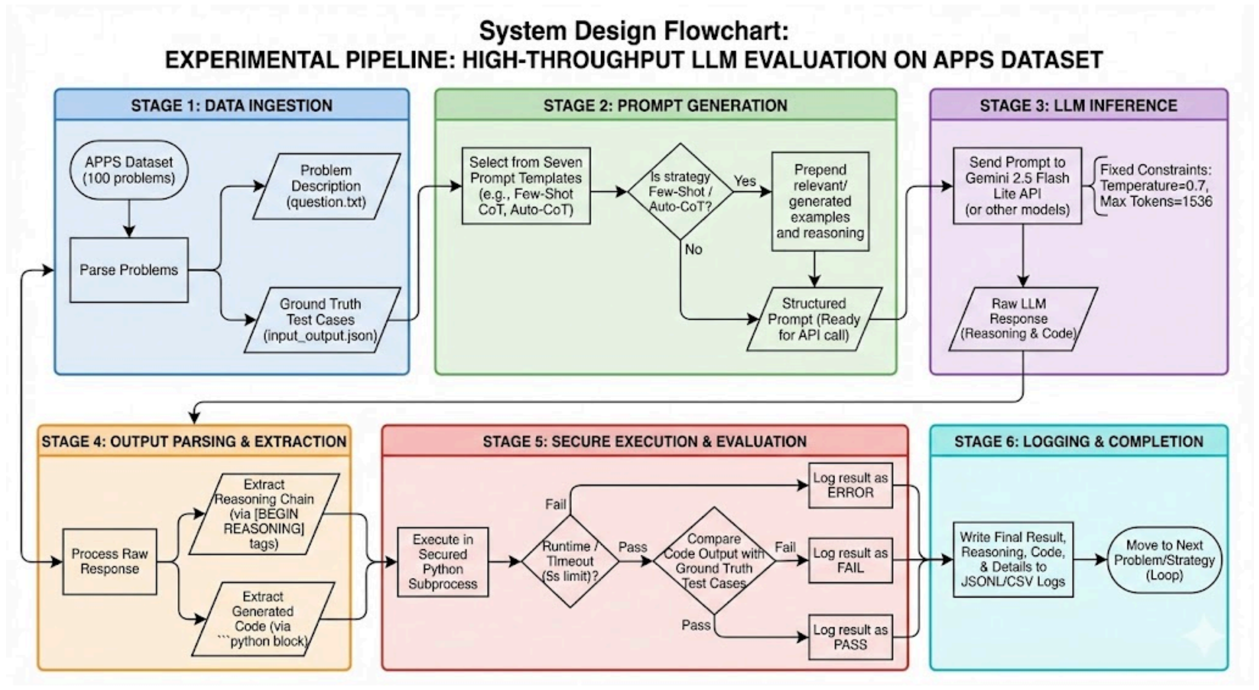
This section details the system architecture, the models utilized, the precise prompting methodology, and the implementation specifics necessary for the study's reproducibility. The methodology systematically isolates the impact of various prompt engineering techniques on code correctness across algorithmic challenges.

System Design and Architecture

The experimental framework employs a structured, six-stage pipeline designed for high-throughput, reproducible testing on algorithmic challenges. The flow ensures strict evaluation and comprehensive data logging.

Architecture Flow

1. Data Parsing: Problem descriptions and ground truth unit tests from the 100 APPS problems are loaded and prepared.
2. Prompt Generation: The problem description is dynamically inserted into one of the seven prompt templates, incorporating strict structural instructions.
3. Model Inference: The structured prompt is sent to the selected LLM API (primarily Gemini).
4. Output Parsing: The raw response is strictly parsed to extract the reasoning chain ([BEGIN REASONING] / [END REASONING]) and the pure Python code (```python block).
5. Code Execution & Evaluation: The extracted code is executed securely in a Python subprocess against the ground truth test cases with a strict 5-second timeout. The result is logged as PASS, FAIL, or ERROR.
6. Logging & Caching: All artifacts (prompt, response, code, and result) are recorded. An Auto-CoT cache stores generated reasoning chains to optimize repeated API calls for training examples.


System Design Flowchart:
EXPERIMENTAL PIPELINE: HIGH-THROUGHPUT LLM EVALUATION ON APPS DATASET
STAGE 1: DATA INGESTION
APPS Dataset (100 problems)
Parse Problems
Problem Description (question.txt)
Ground Truth Test Cases (input_output.json)
STAGE 2: PROMPT GENERATION
Select from Seven Prompt Templates (e.g., Few-Shot CoT, Auto-CoT)
Is strategy Few-Shot / Auto-CoT?
Yes
No
Prepend relevant/ generated examples and reasoning
Structured Prompt (Ready for API call)
STAGE 3: LLM INFERENCE
Send Prompt to Gemini 2.5 Flash Lite API (or other models)
Fixed Constraints: Temperature=0.7, Max Tokens=1536
Raw LLM Response (Reasoning & Code)
STAGE 4: OUTPUT PARSING & EXTRACTION
Process Raw Response
Extract Reasoning Chain (via [BEGIN REASONING] tags)
Extract Generated Code (via ```python block)
STAGE 5: SECURE EXECUTION & EVALUATION
Execute in Secured Python Subprocess
Runtime / Timeout (5s limit)?
Fail
Pass
Compare Code Output with Ground Truth Test Cases
Fail
Pass
Log result as ERROR
Log result as FAIL
Log result as PASS
STAGE 6: LOGGING & COMPLETION
Write Final Result, Reasoning, Code, & Details to JSONL/CSV Logs
Move to Next Problem/Strategy (Loop)

Models Used and Rationale

The project utilized two primary models for evaluation, with one selected for its superior stability in high-throughput testing.

Model	**Platform / API**	**Role in Project**

Gemini 2.5 Flash Lite	Google Generative AI SDK	Primary Evaluation Model. Selected for stability, reliability, and cost-effectiveness in running all 7 prompting strategies across the 100 APPS problems.
Llama 4	Ollama (model="llama4")	Secondary Comparison Model. Used as an open-source comparison to assess the generality of prompting strategies.
Mistral Small Latest	Mistral API	Exploratory Model. Abandoned early due to API stability and throttling issues, limiting the scope of commercial model comparison.

Prompting Strategy Details

The pipeline implemented seven distinct strategies to isolate the impact of reasoning guidance (CoT) versus in-context learning (Few-Shot). All templates strictly set the model's persona to an "expert competitive programmer" and demand a precise output structure for successful automated parsing.

Prompt Strategy	**Purpose**	**Structural Details**
Zero-Shot (create_zero_shot_prompt)	To assess the model's inherent ability to solve problems when explicitly mandated to generate a detailed reasoning structure, without any in-context examples.	Mandates specific reasoning steps: **Restate the problem, Describe optimal algorithm, Give complexity, List edge cases.** Output order: Reasoning -> Code.
Zero-Shot CoT (create_zero_shot_cot_prompt)	Similar to Zero-Shot but uses a slightly different, more generalized set of mandated reasoning steps, including **Identify constraints** and **Explain potential pitfalls**, serving as a comparison for CoT effectiveness based on instruction phrasing.	Mandates specific reasoning steps: **Restate the problem, Identify constraints, Derive optimal solution, Check edge cases, Explain potential pitfalls.** Output order: Reasoning -> Code.
Analytic CoT (create_analytic_cot_prompt)	A much simpler, less-structured prompt to test the model's performance when reasoning steps are merely requested rather than strictly templated.	Requests **Problem analysis, Algorithm design,** and **Edge cases**, followed by the code block. This tests prompt sensitivity to strict formatting.
Few-Shot (create_few_shot_prompt)	To evaluate the benefit of in-context learning based purely on **solution examples**, isolating it from the influence of reasoning.	Prepends **three manually selected examples** (problem/solution pairs) before the target problem. The examples **DO NOT** contain reasoning chains.

Few-Shot CoT (create_few_shot_cot_prompt)	To assess the maximum benefit of providing the model with both **correct code examples** and **high-quality, manual reasoning chains** (Gold Standard CoT).	Prepends **three manual examples**, where each example includes a full, high-quality **[BEGIN REASONING]** block followed by the corresponding **```python** solution.
Random-Few-Shot CoT (create_random_few_shot_cot_prompt**)**	Serves as a critical **control** to differentiate the benefit of *domain-relevant* reasoning from the general effect of a longer, context-rich prompt.	Prepends **three randomly sampled problems** (problem/reasoning/solution). Crucially, the reasoning is non-specific (or derived from the MANUAL_COTS pool, which is often irrelevant to the random problem).
Auto-Few-Shot CoT (create_auto_few_shot_cot_prompt**)**	To evaluate the scalability and efficacy of **self-generated reasoning** in a Few-Shot context. This tests whether "good enough" LLM-generated reasoning is sufficient to prime the model.	Uses a two-step process: **1)** The LLM first generates reasoning (via generate_auto_cot_for_problem) for the three training examples, which is then **cached**. **2)** These examples (LLM-reasoning + human-solution) are prepended to the target problem.

Implementation and Hyperparameters

The study employed fixed hyperparameters across all runs to ensure that any performance variance was attributable solely to the prompt engineering strategy.

Parameter	**Value**	**Rationale**
Temperature	0.7	Balances deterministic correctness (low T) with creative algorithmic exploration (high T).
Top P	0.95	Ensures diversity in token selection while maintaining quality and coherence for code generation.
Max Output Tokens	1536	Set high enough to accommodate the verbose CoT reasoning and the potentially long code solution.
Code Execution	Python Subprocess w/ 5s Timeout	Executes generated code securely and objectively against APPS test cases, preventing infinite loops.

4. Final Results and Analysis

This section synthesizes the findings from the comparative analysis of Llama 4 and Gemini 2.5 Flash Lite models across seven distinct prompting strategies. The goal is to provide a holistic understanding of system behavior and draw key conclusions regarding model effectiveness in solving algorithmic problems.

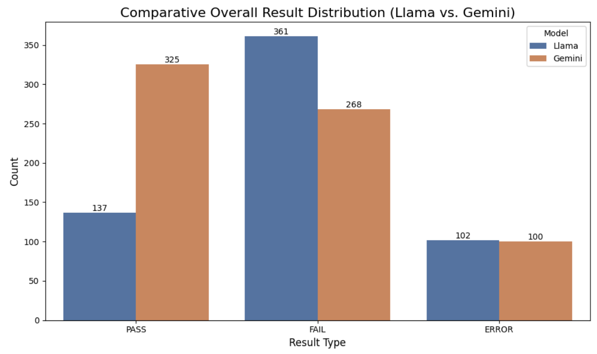
Quantitative Results: Overall Performance and Accuracy

The evaluation reveals a substantial performance disparity between the two models and a non-linear relationship between prompt complexity and code correctness.

1. Model Head-to-Head Comparison

The **Gemini 2.5 Flash Lite** model consistently outperformed Llama 4 across all comparable strategies. Gemini achieved a much higher rate of successful (**PASS**) solutions and a significantly lower rate of logical failures (**FAIL**).

Metric	**Gemini 2.5 Flash Lite (Total)**	**Llama 4 (Total)**	**Interpretation**
Total PASS Results	**325**	137	Gemini is generally more capable of generating logically correct solutions.
Total FAIL Results (Logic Mismatch)	268	**361**	Llama struggled significantly more with logical correctness.
Total ERROR Results	100	102	Both models exhibited similar rates of runtime instability.

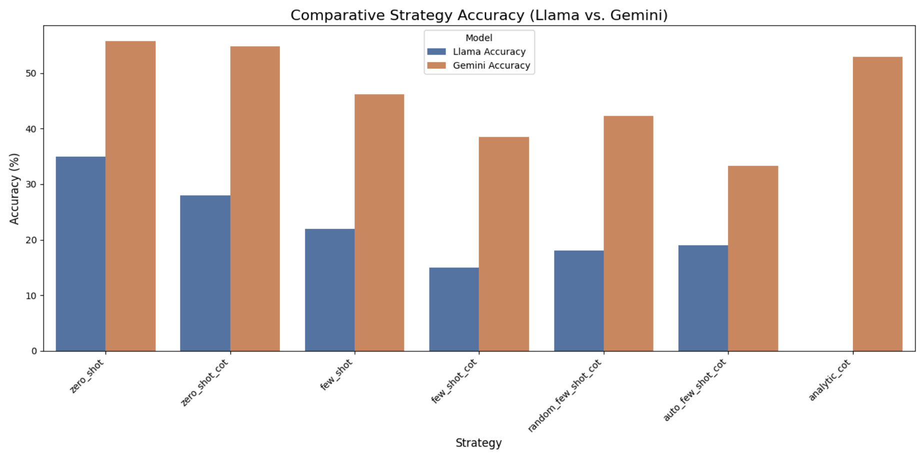


2. Accuracy by Prompting Strategy

The table below details the Pass Rate (Accuracy) for each strategy on the 100 APPS problems.

	zero_shot	zero_shot_cot	few_shot	few_shot_cot	random_few_shot_cot	auto_few_shot_cot	analytic_cot

Llama Accuracy (%)	35.00	28.00	22.00	15.00	18.00	19.00	N/A
Gemini Accuracy (%)	55.77	54.81	46.15	38.46	42.31	33.33	52.88



Detailed Strategy Analysis and Interpretation

1. Gemini: Simplicity Outperforms Complexity

- **Top Performers:** The top strategies for Gemini were the simplest, non-Few-Shot variants: **Zero-Shot (55.77%), Zero-Shot CoT (54.81%),** and **Analytic CoT (52.88%)**.
- **Interpretation:** For the Gemini model, providing direct instructions and demanding a structured analytical path without explicit, lengthy examples proves most effective. Complex Few-Shot strategies—especially auto_few_shot_cot (33.33%)—introduced context size issues and parsing variability, actively degrading performance.

2. Llama 4: Low Baseline, Limited CoT Benefit

- **Baseline Performance:** Llama's highest accuracy (zero_shot at 35.00%) was lower than almost all Gemini strategies, indicating a weaker foundational capability for this task domain.
- **CoT Impact:** Unlike Gemini, the Chain-of-Thought variants did **not** provide a clear, consistent accuracy advantage for Llama 4. The model struggled severely with Few-Shot variants, as exemplified by few_shot_cot (15.00%), suggesting that the added complexity of CoT did not translate into better code generation.
- **Failure Pattern:** Llama's few-shot variants were dominated by **FAIL** results (e.g., few_shot had 67 FAILs vs. 22 PASSs), demonstrating a **persistent struggle with logical correctness** despite successfully generating runnable code.

Common Failure and Error Patterns

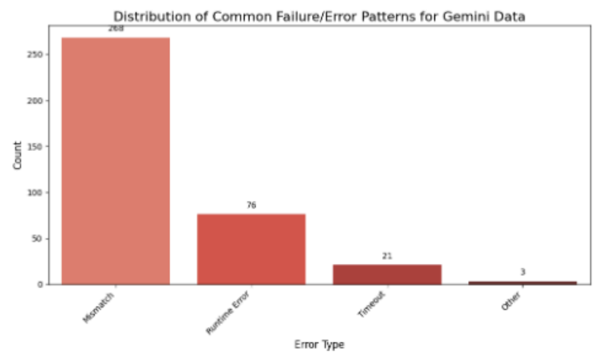
Analyzing the distribution of non-PASS outcomes provides insight into where the models struggle most:

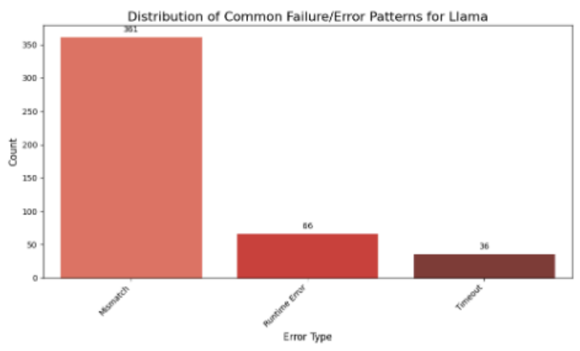
1. Logic Mismatch (FAIL)

- **Prevalence: Mismatch** (logical correctness failure) was the overwhelming primary cause of non-PASS outcomes for **both models** (Llama: 361 occurrences; Gemini: 268 occurrences).
- **Interpretation:** This confirms that the principal challenge lies in **algorithmic derivation and implementation accuracy**, not in syntax or simple execution. The models can generate syntactically valid code, but the logic fails to satisfy the objective test cases.

2. Runtime and Instability

- **Runtime Error (76 for Gemini, 66 for Llama):** Indicates execution problems like exceptions, incorrect input handling, or logical flaws preventing successful completion.
- **Timeout (36 for Llama, 21 for Gemini):** Suggests efficiency issues, where the generated code is correct but too slow for the 5-second limit.
- **Instability in Auto-CoT:** The auto_few_shot_cot strategy was particularly unstable for Gemini, showing a **39% ERROR rate** (27 errors out of 69 attempts), highlighting the compounding instability introduced by involving the model in generating its own training context.





Qualitative Example Insight

A crucial observation was the reasoning output from successful **Zero-Shot CoT** runs on Gemini. For a complex problem (e.g., related to maximum subarray sum or dynamic programming), Gemini often produced concise, correct algorithmic reasoning like:

> "To find the optimal solution, I must use dynamic programming with a sliding window approach. I will maintain a prefix array to calculate sums in O(1) time..."

This indicates that the **structure of the CoT prompt successfully compels Gemini to articulate the optimal algorithm**, even if the final implementation sometimes suffers from minor bugs (like index handling). This strong structured reasoning is the likely reason the simple CoT strategies performed best for Gemini.

5. Reflection, Limitations, & Future Work

This section synthesizes the project's key learnings, acknowledges the constraints that shaped the findings, and outlines avenues for future research.

Reflection and Key Learning

The most significant finding was that **simpler prompting strategies consistently outperformed complex Few-Shot CoT variants** for the high-performing Gemini model. This demonstrates the principle of **prompt parsimony**: relying on the model's inherent, structured reasoning ability (Zero-Shot CoT) is more robust than introducing excessive, potentially error-prone context (Auto-Few-Shot CoT). This led to superior accuracy and stability, challenging the initial hypothesis that more context always yields better results in complex code generation.

Limitations

The study's conclusions are constrained by the following factors:

- **Logic Errors and Instability:** The persistent dominance of **Mismatch (logic failures)** across both models and the high **ERROR** rate in the auto_few_shot_cot strategy indicate a fundamental struggle to synthesize correct solutions, regardless of prompt structure. The observed API instability of Mistral and Gemini during initial testing further limited model diversity.
- **Evaluation Constraints:** The reliance on a strict **5-second timeout** and a closed execution environment led to performance issues related to **code efficiency (Timeout errors)**, indicating that the model often generated correct but unoptimized solutions.
- **Dataset Scope:** The evaluation was limited to **100 "Interview" difficulty problems**, restricting the generalizability of results across different problem categories and languages.

Future Work

Future work should prioritize addressing the observed failure modes:

- **Enhancing Logical Correctness (Mismatch Resolution):** Focus on advanced techniques like **Self-Correction (Self-Refine)** mechanisms, where the model uses unit test feedback to iteratively debug its generated code, directly targeting the primary "Mismatch" error cause.
- **Model Fine-Tuning for Domain Specificity:** A significant, long-term extension would involve **fine-tuning** a base model (e.g., Llama 4) directly on a curated APPS subset. **This crucial step was restricted in the current study due to computational resource constraints.** Fine-tuning is expected to drastically improve logical correctness and output reliability by embedding task-specific code patterns into the model's weights.
- **Model-Specific Optimization:** Conduct tailored searches for optimal prompt structure and hyperparameters for the Llama model, as the current strategies failed to unlock its full potential, underscoring that optimal prompt engineering is highly model-dependent.