# Could LLMs detect and address Code Inefficiencies?

Harish Krishnakumar (924142329), Bismanpal Singh Anand (924176094), Barry Menglong Yao (924332812), Billy Ouattara (920603707), Vrushali Harane (924166376)

## 1 Introduction & Motivation

### 1.1 Motivation

LeetCode-style programming problems are widely used in software engineering interviews and serve as a crucial screening tool for companies ranging from startups to large enterprises, hiring candidates from interns to senior engineers. These problems often involve tasks where candidates are asked to write efficient code and optimize solutions, such as refining an algorithm from $O(n^2)$ to $O(\log n)$. This raises the question: can Large Language Models (LLMs) step into the picture and mimic an engineer's thought process, starting with an inefficient solution and working their way to an optimal one?

This study aims to explore how well LLMs can detect and reason about code inefficiencies, as well as how different prompt engineering techniques can guide these models in generating optimized solutions. Additionally, it will assess which LLMs are most capable of effectively identifying and refining inefficient code, considering factors such as static and dynamic code analysis. We also seek to investigate whether improvements in code efficiency correlate with measurable changes in code quality, including readability, maintainability, and performance. By applying human-inspired techniques like reasoning and self-feedback through prompting, this work explores how LLMs can be guided to produce more efficient and readable code solutions.

### 1.2 Theory

Code inefficiency includes unnecessary memory consumption and inefficient execution. As defined in [18], inefficiencies can be categorized into two types: (1) memory inefficiency, such as space inefficiencies or for example memory leaks caused by holding references to unused objects, and (2) execution inefficiency, arising from unnecessary operations like inefficient algorithms or protocols. Benchmarking the LLMs to evaluate the trade-offs LLMs make between code efficiency, correctness, and memory optimization, not only at a semantic level but also according syntactic structure is important.[12] [13]

### 1.3 Research Questions

In this paper, our objective is to specifically address these research questions: **RQ 1:** How accurately LLMs can detect and reason code inefficiencies on LLM? **RQ 2:** How can we use different prompt engineering techniques on LLM models to produce efficient code? **RQ 3:** Which LLM model are suitable to identify and refine the inefficient code?[1]

### 1.4 Contributions

Our work has the following contributions: (1) We benchmark LLMs' capabilities on identifying and refining inefficient codes. (2) We propose two noval code refinement methods, Reasoning-based Method and Self-feedback based Method, which further increase the efficiency of generated codes.

[1]Our code is publicly available on `https://github.com/harish876/CodeRefineAI`

## 2 Background

### 2.1 Literature Review

Earlier studies have explored various approaches for analysis and improvement of a code's efficiency through the use of static and dynamic analysis techniques. Mercury [2] and Enamel [10] proposes a tailored benchmark for measuring code efficiency in LLM-generated programs by enabling a structured dataset for evaluating memory and execution inefficiencies along with functional correctness. CrystalBLEU [3] proposes a similarity metric used for code and can distinguish similar and dissimilar programs better than existing approaches. CodeBLEU [12]introduces a syntactic and semantic way of assessing code generation quality. In contrast, Tristan et al [1] benchmarks the efficiency of codes generated by LLMs. Ecco [17] investigates several prompt engineering strategies on LLM to generate optimized without sacrificing functional correctness, a paramount consideration for our study.

However, there is no combined study which evaluates LLM's [6, 15, 4] generated code both statically and dynamically and also experiment with different prompt engineering techniques with different LLMs for efficient code generation. Using inspirations from these research works, our study seeks to benchmark a variety of LLMs on their ability to identify and ameliorate inefficiencies using prompt engineering techniques and execution-centered evaluations.

### 2.2 Novelty

Our work introduces a new approach to improve LLM-generated code by using different prompting techniques. First, we use self-feedback within the same model to refine its output. Second, we generate reasoning of the code being inefficient, to provide additional guidance to the LLM. We also create a knowledge graph to visualize each question linked to it's inefficiency and reasoning. We then combine all these techniques with runtime metrics and code similarity metrics to evaluate how efficiently the LLM generates code. This approach brings together feedback loops and structured reasoning to help the model produce better, more efficient code compared to previous methods that focused on isolated techniques like Reflexion for self-improvement [11], prompt chaining [18], or fine-tuning on LeetCode datasets [17]. By integrating these methods, we aim to create a more comprehensive and effective pipeline for optimizing LLM-generated code.

## 3 Methodology

### 3.1 Dataset

For our project, we chose to proceed with the **Venus** dataset, as outlined in our proposal. The structure of this dataset makes it particularly suitable for our objectives, as it contains LeetCode problems along with solution submissions from multiple candidates. The variability in these submissions reflects the diverse approaches that can be taken to solve a given problem. Each problem in the dataset includes two distinct sets of candidate solutions, one based on runtime and the other on memory usage. Additionally, each entry includes the corresponding code and test cases, enabling effective evaluation of the solutions. The **Venus** dataset comprises a total of $2.81k$ entries. However, to mitigate the bottlenecks associated with querying and fine-tuning large language models (LLMs), we reduced the dataset to a curated subset of 200 samples. This subset was selected to maximize diversity across problem types and topics, thereby preserving representativeness while significantly reducing computational overhead. Each sample contained problems spanning various data structures and algorithmic concepts. However, our deliberate selection of diverse-topic problems, rather than single-topic-focused problems, might have introduced bias. Bias in the sense of disproportionately favoring problems involving multiple topics. In total, the 200 samples span about 19 data structures and 30 algorithm concepts.

## 3.2 Datatset Utilisation

Our goal is to assess the ability of large language models (LLMs) to detect and improve code inefficiencies based on runtime and memory performance. To support this objective, we performed pre-processing on the Venus dataset to restructure the data in a way that facilitates efficient analysis. To enable the distinction between efficient and inefficient code, as well as to measure improvements from one version to another, we constructed a simple classifier. For each set of submissions, runtime-based and memory-based, we analyzed their respective performance distributions. Submissions falling within the 25th percentile were labeled as *efficient*, those within the 75th percentile were labeled as *inefficient*, and submissions falling between the two thresholds were labeled as *moderately efficient*. In our 200-sample subset, each problem contains, on average, approximately 13.51 runtime-based submissions per category (efficient, moderately efficient, and inefficient). For memory-based submissions, we observed an average of 5.55 submissions per category. Each problem also has an average of 128 test cases and 66 easy, medium, and hard questions. We used boilerplate code from the data to generate a code template, which is dynamically injected with test cases and run for each program.

## 3.3 Evaluation Tools

We built a code execution client, coderefineai-executor, that runs Python code in a sandboxed environment. This tool is built on top of Judge0, a code execution platform that supports multiple languages, including Python 3.7, which we used in our experiment. We opted for a self-hosted Judge0 instance since the cloud offering imposed a rate limit of 50 submissions per day, whereas our team required a throughput of over 200 submissions daily.

The code execution client provides features for submitting code, retrieving submission details, and polling for a terminal status: **Accepted, Wrong Answer, Runtime Error (NZEC), or Time Limit Exceeded**. It also allows executing code samples $K$ times and generates execution results, including: (1) **Functional correctness status**: Accepted, Wrong Answer, Runtime Error (NZEC), or Time Limit Exceeded. (2) **Runtime** (in seconds). (3) **Memory usage** (in KB). (4) **Error stream** (from stderr, if the execution fails).

To facilitate large-scale testing, we developed a **parameterized Python script** that standardizes execution and result collection. The script has the following functions: (1) Accepts a **parent directory** for fetching and storing results. (2) Takes a **JSON file** with generated solutions. (3) Provides an option to **select solutions** from either the original dataset or an LLM-generated solutions file.

This script enabled us to generate labeled result datasets. To ensure consistency, we passed both reference solutions and LLM-generated code through our execution framework to homogenize runtime and memory metrics. To avoid discrepancies, we ignored the runtime and memory metrics from the original dataset during evaluation. Additionally, we developed a Evaluation Pipeline with commonly used utility functions, abstracting repetitive tasks to assist with analysis. Team members rely on these utilities to streamline their evaluations.

## 3.4 Method for RQ1: Identify Inefficient Code

In this section, we address our first research question: How accurate are LLMs in identifying memory and runtime inefficiencies? To answer it, we design experiments to assess whether LLMs can classify efficient codes as efficient and inefficient codes as inefficient, thus converting our problem into one binary classification task. Specifically, each leetcode problem $p$ in our test dataset is associated with a set of efficient solutions $[s^e_1, s^e_2, ...,]$ and a set of inefficient solutions $[s^n_1, s^n_2, ...,]$[2] with respect to one efficiency metric $m$, such as runtime or memory usage. We then randomly sample one solution $(s_i, y_i)$ from these two solution sets, where $s_i$ is the solution code and $y_i$ is the corresponding label where 0 stands for

[2]Note that we assess LLM's capability in two types of efficiency, a.k.a., memory efficiency and runtime efficiency, separately. But we adopt a unified notation to represent both types of efficiency for simplify.

inefficient code and 1 stands for efficient code. We then feed $p$, $s_i$, and $m$ into one LLM and trigger it to predict the efficiency of $s_i$ with the prompt "Given the leetcode problem $\{p\}$ and one solution $\{s_i\}$, check whether the solution is efficient or not regarding to $\{m\}$. Write Yes if it is correct, and No if it is not". We finally parse LLM's prediction as $\hat{y}_i$, where 0 and 1 stands as inefficient and efficient code, respectively.

Due to computational resource limitation, we conduct our experiments on a sampled test dataset, consisting of 200 leetcode problems as detailed in Section 3.1. For each leetcode problem and each efficiency type, a.k.a., runtime or memory usage, we randomly sample one efficient solution and one inefficient solution from its corresponding efficient and inefficient solution lists. We leverage LLaMA-3.2 [4], the leading open-source generalist LLM, in our experiments. Since we run experiments on Colab notebook[3] with limited computational resources, we use the LLaMA-3.2-1B model[4].

## 3.5 Method for RQ2: Refine Inefficient Code

In this section we try to answer the question: How can we use different prompt engineering techniques on LLM models to remove inefficiences from the code? We applied this techniques to 200 sample problems have equal Easy, Medium and Hard problems. For code execution we used self-hosted remote sandbox environment, Judge0, which allows for accurate measurement of functional correctness, runtime, and memory usage. Code similarity is assessed using CodeBLEU [12] as a preliminary step to evaluate how closely the LLM-generated code matches reference implementations. We used 2 LLM models to run these prompt engineering techniques. Firstly, open source **Meta Llama3.2-1B** and secondly, closed source **Google Gemini 2.0 Flash**. We decided to work with Meta Llama3.2-1B for its agentic retrieval and summarization tasks. It could generate reasoning and natural language feedback along with optimized code. We chose Meta Llama3.2-1B and not a higher version of Llama because we were restricted on the infrastructure. We did not have access to A100 GPU. The Llama3.2-1B was able to generate code in proper formatting enclosed in a Solution class. But, there was scope of improvement in improving runtime and memory efficiency. We decided to use Gemini-2.0-flash which is a powerful, low-latency model with enhanced performance and can generate more efficient codes as compared to the previous models. It is also accessible through API, this helped us to overcome our infra limitations.

### 3.5.1 Vanilla Prompting

The first method for our experimentation was Basic Prompting/ Vanilla Prompting. This technique is defined as directly providing LLM with a query, without any kind of engineering around it to try to improve the response of LLM's performance[16]. We prompt Meta Llama3.2-1B in a very basic manner. We simple send in the inefficient code block and asking the LLM to optimize it. We also ensure that we get back the same functional call to align with our evaluation pipeline. The exact prompt can be referred to from Appendix

### 3.5.2 Reasoning-based Method

The second approach came from a thought, "What if the model knows the cause of inefficiency?". Thus, an attempt was made to let the model know the reasoning behind provided code being inefficient. For this, we utilized Gemini-2.0-flash to generate an analysis on the code and provide an explanation/ reasoning for the code being inefficient. We chose this model because even though it's parameter size is not revealed officially, it is recognized as a large scale AI model. The prompt utilized for this task is :

```
---------------
The following Python code seems to be inefficient:
```

[3]https://colab.research.google.com/

[4]https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct

```
        ‘‘‘python
        {inefficient_code}
        ‘‘‘
        Reason for inefficiency:
        {reasoning}

    Please rewrite the code based on the provided reason, and only return the
    corrected code, no explanation. Use Python 3.8 syntax, and ensure the entire
    solution is enclosed in a ‘class Solution‘, maintaining the same function
    name as in the input code. Directly start with the code which can be put
    in the executor and no statement or word before that.
    ---------------
```

This gave us an analysis over why the code is inefficient which could now be used to guide the smaller, 1B parameter model- Meta Llama to help in optimization. This process of using a larger model to 'teach' a smaller model is known as 'distillation'[14]. Something similar is done here in an attempt to enhance the code generation of a smaller LLM. The Meta Llama is then prompted to perform optimization on the inefficient code, but this time, along with an explanation to why the code is ineffective. The prompt for this task can be referred from appendix.

This prompt, finally provides an inefficient code along with an explanation of the inefficiency as context to the model. The LLM then reviews these together to optimize the code and return back a potentially better code which is expected to have a better runtime and complexity.

#### 3.5.3 Self-feedback based Method

The third approach we tried is inspired from Reflexion feedback paradigm [11]. We did experimentation based on the 'NL+Exec Refine' method mentioned in [17]. It comes from the ideology - 'Can model improve itself if we give execution results for it's own generated code to itself for few iteraations?'. Specifically, we first obtain the execution results on the first optimized code generated by LLM using Judge0, then asked the LLM to write Natural Language feedback on the incorrect/inefficient parts of the generated code. We gave the execution results and feedback generated to the model and asked to generate a refined code. We ran 3 such rounds of refinement iterations. We used the prompts mentioned in A.2 to generate optimized code , natural language feedback and refined code from LLM.

### 3.6 Method for RQ3: Compare LLMs

#### 3.6.1 Compare LLMs on Identifying Inefficient Code

We further extend our method to Gemini [15], a state-of-the-art generalist LLM that excels across most benchmark datasets, for experiments on identifying inefficient code, as detailed in Section 3.4. We compare LLaMA-3.2-1B and Gemini-2.0-Flash [5] on two settings. (1) Across all solutions, which LLM perform better? (2) For solutions under one efficiency type, e.g., runtime or memory usage, which LLM perform better?

#### 3.6.2 Compare LLMs on Refining Inefficient Code

We ran all the 3 prompt engineering methods, discussed in Section 3.5, on both **Meta Llama3.2-1B** and **Gemini-2.0-flash** to address following questions: (1) Which LLM performs better for which

[5] https://deepmind.google/technologies/gemini/flash/

prompt engineering techniques? (2) What are the topics of problems on which it is able to produce efficient code ? (3) How much more runtime efficient code we can produce considering the difficulty level (Easy/Medium/Hard) of the problem?

## 4 Results

### 4.1 Result for RQ1: Identify Inefficient Code

(1) As shown in Table 1, for sampled 778 solutions from the 200 leetcode problems, LLaMA-3.2-1B can correctly identify the efficiency label for 203 out of 778 solutions, which leads to an accuracy of 26.09%, indicating LLaMA's weak capability to identify the inefficient codes. (2) Considering the efficiency type, LLaMA achieves an accuracy of 26.40% on Runtime Efficiency Type and 25.78% on Memory Efficiency Type, indicating that Memory-related efficiency is slightly harder to identify.

| Model | Efficiency Type | Accuracy(%) | Precision(%) | Recall(%) | # Correct | # Total |
|---|---|---|---|---|---|---|
| LLaMA-1B | Runtime | 26.40 | 26.40 | 26.40 | 104 | 394 |
| Gemini-2.0-Flash | Runtime | 39.59 | 39.59 | 39.59 | 156 | 394 |
| LLaMA-1B | Memory | 25.78 | 25.78 | 25.78 | 99 | 384 |
| Gemini-2.0-Flash | Memory | 34.64 | 34.64 | 34.64 | 133 | 384 |
| LLaMA-1B | All | 26.09 | 26.09 | 26.09 | 203 | 778 |
| Gemini-2.0-Flash | All | 37.15 | 37.15 | 37.15 | 289 | 778 |

Table 1: Accuracy on identifying code efficiency.

### 4.2 Result for RQ2: Refine Inefficient Code

Results from each of the prompting method are as follows. All the techniques were applied on 200 problems with a truncated test case set of 50 cases per problem.

#### 4.2.1 Vanilla Prompting

The most basic prompting technique, vanilla prompting gave the results as follows: The model generated functionally correct code for 44.8% of the problems. We got a wrong answer in 27.3% of problems and runtime error for 27.9%. We also compare the run-times of the output code categorized by difficulty level and found out that LLM's result had improved runtime in 93% of easy, 96% of medium and 100% of hard problems which got accepted, compared to the solution provided. The results can be compared from Table 3. This is surprising as it implies that the difficulty level of code does not make it harder for the model to resolve them. However, comparing the topics of questions (like hashmaps, dynamic programming, array etc.), we found out that the LLM could not improve, in comparison to the baseline efficient solution in our dataset on topics like dynamic programming and two pointers. This can be viewed from in appendix.

#### 4.2.2 Reasoning-based Method

The refined approach of reasoning-based prompting worked well and produced significantly better results: The LLM reached functional correctness for 78.8%, an increase of 34% from vanilla prompting. It gave wrong answer for only 9.93% of problems and a runtime error in just 10.6% problems. Based on difficulty, we got 65.78%, 80.85% and 67.64% of the solutions' runtime improved for easy, medium and hard respectively, as shown in Table 3. One thing to note is that even though the percentages seem lower than vanilla prompting, the count is higher here as we have a lot more solutions accepted in this. However, yet again, no trend is observed as easy and hard problems have equivalent outcomes- suggesting that difficulty does not impact the model's performance. Comparing the topics of each problem, we found out that reasoning-based method did improve the solutions for two pointer problems by 33.3% but still failed at dynamic programming. This indicates that model indeed struggles in complex topics like DP as shown in in appendix.

#### 4.2.3 Self-feedback Loop Method

The self-feedback loop's third refinement on the contrary produced results where 60.5% code was functionally correct, which is not as good as reasoning-based method. It gave 6.4% runtime error and 33.2% runtime error. But, it produced more efficient results than any of the above prompting techniques. It was 97.14% faster than the most efficient reference present for Easy problems in the dataset. Similarly, it produced 96.77% and 100% more runtime efficient results for Medium and Hard difficulty level of problem. The model performed well for Leetcode Hard problems as well. As metioned in Table and Table in appendix, it did not perform great for Math and dynamic programming type of problems. We also observed that with each refinement of the self-feedback there was significant improvement in the % of accurate code generated as mentioned in Table 2.

| **Model** | **First Iteration** | **Second Iteration** | **Third Iteration** |
|---|---|---|---|
| Meta Llama3.2-1B | 46.67 | 59.45 | 60.50 |
| Gemini-2.0-flash | 66.8 | 67.8 | 74.50 |

Table 2: % of Accurate Code Produced From The Iterations of Refinement for Self-Feedback Loop

| **Model** | **Category** | **Vanilla Prompting** | **Reasoning-based** | **Self-feedback** |
|---|---|---|---|---|
| Meta Llama3.2-1B | Acceptance (%) | 44.8 | 78.8 | 74.50 |
| Gemini-2.0-flash | Acceptance (%) | 77.9 | 80 | 60.50 |
| Meta Llama3.2-1B | Wrong Answer (%) | 27.3 | 9.93 | 2.1 |
| Gemini-2.0-flash | Wrong Answer (%) | 9.30 | 9.33 | 6.4 |
| Meta Llama3.2-1B | Runtime Error (%) | 27.9 | 10.6 | 23.4 |
| Gemini-2.0-flash | Runtime Error (%) | 12.8 | 10.7 | 33.2 |

Table 3: Results From The Three Prompting Techniques

| **Model** | **Category** | **Vanilla Prompting** | **Reasoning-based** | **Self-feedback** |
|---|---|---|---|---|
| Meta Llama3.2-1B | Easy | 93.1 | 65.78 | 97.14 |
| Gemini-2.0-flash | Easy | 77.77 | 65.78 | 96.87 |
| Meta Llama3.2-1B | Medium | 96.0 | 80.85 | 96.77 |
| Gemini-2.0-flash | Medium | 76.47 | 77.27 | 100.00 |
| Meta Llama3.2-1B | Hard | 100.0 | 67.64 | 100.00 |
| Gemini-2.0-flash | Hard | 80.43 | 83.87 | 96.87 |

Table 4: Results From The Three Prompting Techniques for Runtime Improvement in %

## 4.3 Result for RQ3: Compare LLMs

### 4.3.1 Compare LLMs on Identifying Inefficient Code

(1) As shown in Table 1, on sampled 778 solutions, Gemini-2-Flash achieves an accuracy of 37.15%, significantly outperforms LLaMA-3.2-1B, which is aligned with the fact that LLaMA-3.2-1B is much smaller in model size compared with the Gemini-2-Flash model. (2) Considering the Efficiency Type,

Gemini-2.0-Flash outperforms LLaMA-3.2-1B by 13.19% and 8.86% on Runtime Efficiency Type and Memory Efficiency Type, respectively.

### 4.3.2 Compare LLMs on Refining Inefficient Code

(1) As shown in Table 3, sampled on a data set of size 200, Reasoning-based prompt engineering method with Meta Llama3.2-1B performed the best for generating functionally correct solution i.e 78.8% over all other methods. (2) As shown in Table 4, Meta Llama3.2-1B performed equally well for Easy and Hard difficulty of problems for Vanilla and Reasoning-based prompting. However, Google Gemini 2.0 performed the best for Hard difficulty of problems (3) As shown in Table 4, third iteration/refinement of Self-feedback loop generated the most efficient solutions as compared to other prompt engineering techniques. Google Gemini 2.0 Flash generated faster runtime solutions for Medium problem than Meta Llama3.2-1B.

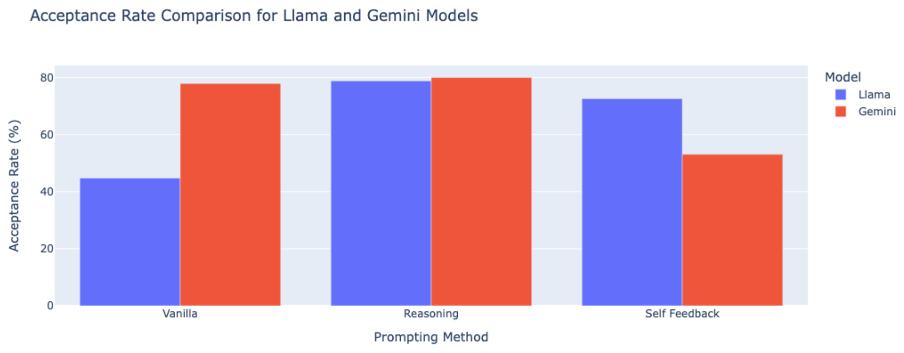

Figure 1: Comparing Acceptance Rates of both LLMs with each prompting technique

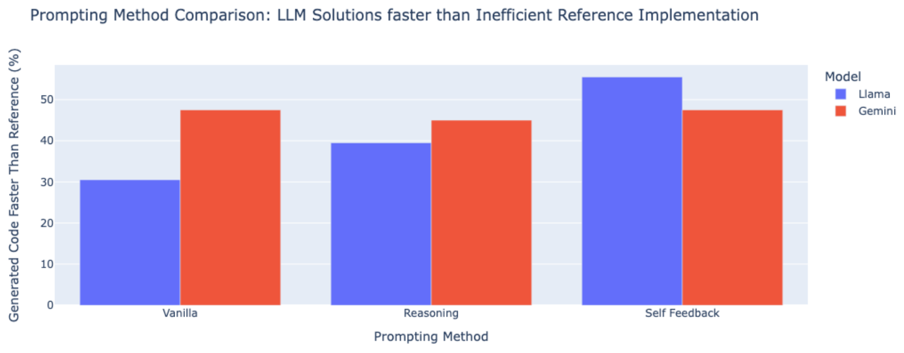

Figure 2: Comparing LLM generated code's runtime solution from the inefficient reference with each prompting technique

## 5 Threats to Validity & Limitation

In this section, we identify and discuss potential limitations and biases that may affect the validity and generalization of our study findings.

- **Compute for running complex models**: Since we did not have access to GPU's and compute to run large language models with a larger parameter size we made a key design decision to use a language model with smaller number of parameters i.e the Meta Llama 3.2 1B and Gemini 2.0 Flash a model with a larger parameter size through Google's Gemini API. Thus, a limitation we address here is the use of a close sourced model in Google Gemini 2.0. With access to better compute, we could have used a open sourced large language model with a bigger parameter size. However this limitation does not compromise our results but masks key details as to why a larger language model exhibits the particular behaviour.

- **Dataset**: We use a truncated dataset of about 200 problems and apply 2 models and 3 prompting techniques each to collect observations and results for our experiments. Since compute was a bottleneck in terms of code generation, we chose to go with a smaller dataset. To mitigate skewness in our data we chose a representative set of problems covering 19 data structures and 30 algorithmic complexity ensuring different variety of questions to be covered.

- **Runtime Metrics**: We use a dataset which provides runtime and other metrics pertaining to the execution of code. While the intent to choose the dataset was to utilize these attributes we acknowledge that these metrics might not be homogenous with the results we run using our code executor. Thus to overcome this limitation, we run the reference solution with our code execution pipeline and augment the datset to provide a more accurate comparision of LLM generated code's performance with the reference implementation.

- **Dataset Selection**: We categorize the problems present in our dataset as efficient and inefficient using a threshold as referenced in 3.2. The choice of the threshold was taken according to standard values used widely in experiments but is subject to bias for our experiments. To overcome this, we benchmark LLM's performance with both categories i.e, efficient and inefficient to generate results. However since this would introduce multiplicity of a factor of 2 to all our experiments, we use the results with reference to inefficient solutions and publish the results of our other findings in our codebase.

- **Prompting Techniques**: We observed variability in behaviour for both Llama and Gemini models where the generated code would sometimes deviate from the intended prompt. We also limit our self feedback loop to 3 iterations due to compute bottleneck and cost. To overcome these limitations we repeat the process for obtaining the code when the intended behaviour is not achieved and change prompts and techniques to prevent LLM's from hallucinations. This gradually led us to choosing the prompting techniques we have used in the experiments resulting in reliability and minimizing outliers in our output.

## 6 Discussion

### 6.1 Principal Outcomes

We explored 3 prompt engineering techniques - Vanilla Prompting, Reasoning Based Prompting and Self Feedback Loop with 3 iterations for 2 types of Language Models. We use Meta LLama 3.2 1B vs Gemini 2.0 Flash. The former being a smaller auto-regressive model vs the latter, a commercial model with a larger parameter size. We summarise our findings in the following points:

- **Functional Correctness**: We observe that Llama's performance improves across different prompting techniques—vanilla, reasoning, and self-feedback—with acceptance rates of 39.7%, 69%, and 72.5%, respectively. This suggests that even a smaller language model benefits significantly from advanced prompt engineering techniques. However, the same trend is not observed in Gemini, where vanilla prompting and reasoning yield similar acceptance rates [6] (68% and 70%), while performance declines with self-feedback to 53%.

[6]Acceptance rate is the percentage of total solutions that are functionally correct, meaning they have passed all test cases in the dataset.

- **Runtime**: We compare the runtime performance of Llama and Gemini across the three prompting techniques relative to the reference solution, using Win Rate as our efficiency metric. Llama's generated code improves in speed with more advanced prompting techniques—vanilla, reasoning, and self-feedback—achieving Win Rates of 30.5%, 39.5%, and 55.5%, respectively. This indicates that smaller language models benefit from prompt engineering. However, Gemini does not exhibit the same trend, with its Win Rate [7] remaining relatively stable between 45% and 47% across all three prompting techniques.

- **Code Similarity**: We use CodeBLEU [12] as a code similarity metric, as it provides a more representative measure than natural language-based techniques like cosine similarity by considering both syntax and data flow. We observe that for Llama, code similarity starts high—closely matching the inefficient reference solution—but decreases as more advanced prompting techniques like reasoning and self-feedback are applied. This trend aligns with the observed improvements in runtime efficiency. In contrast, Gemini consistently produces a lower CodeBLEU score, suggesting it constructs solutions from scratch rather than refining existing ones. However, this trend does not translate into a clear, linear improvement in runtime efficiency.

- **Code Quality**: We evaluate the quality of generated code using the Maintainability Index[9], Cyclomatic Complexity [8] , and Halstead's Volume[5]. We observe that code generated using vanilla prompting and reasoning scores poorly across these metrics. However, self-feedback significantly improves code quality, with self-feedback-generated code outperforming both vanilla and reasoning approaches for both models.

From these findings, we answer our research question as follows:

**RQ 1: How accurately LLMs can detect and reason code inefficiencies on LLM?**

The current freely available LLMs are still weak on identifying code efficiency, given that LLaMA-3.2-1B and Gemini-2.0-Flash achieve an accuracy of 26.09% and 37.15%, respectively, as discussed in Section 4.3.1.

**RQ 2: How can we use different prompt engineering techniques on LLM models to produce efficient code?**

Providing more context in the form of reasoning and iterative self feedback improves code generation in language models with smaller parameters whereas the results are not pronounced in Large Language models. We see an improvement from 30.5% efficient solutions to 55.5% efficient solutions from vanilla to self feedback in Llama whereas for Gemini the increase is negligible and regresses for reasoning (45%) from vanilla prompting (47.5%). Thus, iteration and added context benefits small language models more.

**RQ 3: Which LLM model is suitable to identify and refine the inefficient code?**

As shown in our results, while Gemini-2.0-Flash is better in identifying inefficient codes, Llama provides a better result in refining inefficient codes in terms of code efficiency and code quality. Llama responds well to advanced prompting techniques with an improvement from 30.5% to 55.5% from vanilla to self-feedback in runtime. We also observe improvement of Llama over Gemini in all aspects of code quality. Refer to Appendix 7, Table 7 for a comparison of code quality metrics. Thus we report that a smaller language model when provided more context distilled from a large model or feedback from itself, performs better.

[7] *Win rate* is defined as the percentage of instances where elements in $\mathbf{arr}_1$ are strictly less than their corresponding elements in $\mathbf{arr}_2$, given by:

$$\text{Win Rate} = \left( \frac{\sum_{i=1}^{n} 1(a_{1,i} < a_{2,i})}{n} \right) \times 100$$

where $1(\cdot)$ is the indicator function that returns 1 if the condition inside holds true and 0 otherwise, and $n$ is the total number of comparisons.

### 6.2 Comparison to Prior Studies

In alignment with work done in *Ecco*[17], self-feedback loop prompting technique was applied, where we give natural language as well as executor results feedback to LLM to refine code, produced much more runtime efficient/faster code than other prompt engineering techniques. However, reasoning-based method generated more functionally correct code. Due to smaller context window of open-source LLMs we changed the prompt mentioned in *Ecco*[17] to generate feedback restricted within 20-30 words.

We perform a static analysis of the generated code by evaluating its similarity to the reference solution and calculating code complexity metrics such as Maintainability Index [9], Cyclomatic Complexity [8], and Halstead's Volume [5]. This approach provides a comprehensive view of code quality, highlighting the impact of different prompting techniques on both the structural integrity and efficiency of the code. By comparing these metrics across various LLM models, we aim to explore the relationship between code optimization and its static characteristics. Unlike previous work, our study uniquely investigates both the efficiency of the code and its static properties, offering a holistic assessment of the generated solutions.

### 6.3 Future Work

- **Scale to Larger Datasets and Models**: Due to computational resource limitations, our current project focuses on a small dataset with around 800 code samples and two smaller models like LLaMA-3.2-1B and Gemini-2-Flash. With more computational resource and budgets, we can extend our experiments to larger models like LLaMA-3.3-70B [8] and GPT-4o [6].

- **Fine-tuning based Methods**: Due to computational resource limitations, we focus on prompt-based methods, while we assume that further fine-tuning LLMs on our tasks will achieve a better performance.

### 6.4 Conclusion

In this study, we evaluated the ability of large language models (LLMs) to detect and optimize code inefficiencies, focusing on runtime and memory usage as well as semantic and syntactic quality through varied prompt engineering methods. Our experiments indicate that current LLMs still face significant challenges in accurately identifying inefficiencies. Nonetheless, prompt engineering techniques, such as reasoning-based prompting and iterative self-feedback, significantly enhanced the functional correctness and runtime efficiency of the smaller model(LLaMA-3.2-1B). These findings highlight the importance of advanced prompting strategies in improving code generation efficiency. However, limitations stemming from the use of LeetCode-style algorithmic challenges suggest the need for further validation with real-world software development scenarios. Future research should explore more diverse datasets and consider fine-tuning approaches to further enhance the practical applicability of LLM-generated code.

## 7 Team Membership and Attestation

**Harish:** Led the development of the execution client and platform for the project, contributing to the creation of code submission pipelines that facilitated the execution of code generated by LLMs. In collaboration with Bismanpal, played a key role in building a result evaluation pipeline to analyze outcomes from various prompting techniques and experiments. Contributions included developing utility functions, conceptualizing and implementing evaluation statistics for effective presentation, and integrating both static and dynamic code analysis. This integration leveraged an executor, along with code similarity and complexity metrics, to enhance the evaluation process.

**Bismanpal:** Developed **Experiment One: Vanilla Prompting on Inefficient Code** and **Experiment Two: Reasoning-based Refinement on Inefficient Code** for **RQ2**, focusing on refining code

[8] `https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct`

through prompting-based techniques. Initiated the development of a **Knowledge Graph** to facilitate simpler visualization which subsequently led to the ideation of the reasoning-based prompting method. Additionally, contributed to the results pipeline, specifically the visualization of results across models and prompting techniques, utilizing interactive plotting libraries.

**Barry:** (1) Address **RQ1**, which involves developing a codebase that locally prompts LLaMA and interfaces with the Gemini API to evaluate runtime and memory efficiency, running RQ1 experiments with LLaMA and Gemini for around 800 solutions, analyzing experiment results and writing report. (2) Coordinate project management activities including scheduling project meetings, facilitating discussions in the meeting, and coordinating task assignments.

**Billy:** Responsible for the **Dataset** section, including identifying, preprocessing, and tailoring the Venus_t dataset to align with our project goals. Helped **Bismanpal** in leveraging LLMs to identify reasons for code inefficiencies and use them to construct the knowledge graph.

**Vrushali:** Developed **Experiment Three**: **Self-Feedback Loop for Code Refinement** for **RQ2**. Experimented with different feedback as well as refinement prompts to deal with the problem of restricted input tokens to LLMs considering limited infrastructure. Integrated the Executor developed by Harish into the self-feedback loop prompting and designed the prompt to include execution results as well.

## Code & Data Availability Statement

Our code and data samples are publicly available on `https://github.com/harish876/CodeRefineAI`. The original dataset is obtained from `Venus_t`, available at dataset. For execution of code and providing dynamic runtime data we use our tool executor. For evaluating code similarity, we use the CodeBLEU library, and for measuring code complexity, we utilize the Radon library. To interact with multiple Large Language Models (LLMs) through a unified interface, we rely on Langchain and a set of necessary abstractions. For local inference, we run models using Huggingface and vllm [7] on Google Colab with GPU support. We also use Google's free Gemini API to interface with the Gemini 2.0 Flash model. All results published in this report can be replicated in this notebook present in our repository. This is the analysis pipeline used across all our experiments. Instructions on how to run the scripts and where to find the dataset files are present in the readme.

## References

[1] T. Coignion, 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*, EASE 2024, page 79–89. ACM, June 2024.

[2] M. Du, A. T. Luu, B. Ji, Q. Liu, and S.-K. Ng. Mercury: A code efficiency benchmark for code large language models. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[3] A. Eghbali and M. Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[4] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[5] M. H. Halstead. *Elements of Software Science.* Elsevier, 1977.

[6] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[7] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[8] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[9] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 337–344, 1992.

[10] R. Qiu, W. W. Zeng, J. Ezick, C. Lott, and H. Tong. How efficient is llm-generated code? a rigorous high-standard benchmark, 2025.

[11] N. Shinn, S. Yao, and D. Ziegler. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366v4*, 2023.

[12] L. M. Z. J. Shuai Lu, Daya Guo and L. Zhang. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[13] T. sitter Contributors. Tree-sitter: An incremental parsing system for code analysis, 2025. Accessed: 2025-01-19.

[14] Z. Sun, C. Lyu, B. Li, Y. Wan, H. Zhang, G. Li, and Z. Jin. Enhancing code generation performance of smaller models by distilling the reasoning ability of llms, 2024.

[15] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[16] S. Vatsal and H. Dubey. A survey of prompt engineering methods in large language models for different nlp tasks, 2024.

[17] S. Waghjale, V. Veerendranath, Z. Z. Wang, and D. Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024.

[18] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 421–426, 2010.

## A Appendix:

### A.1 LLM Usage Details

- We used Meta Llama3.2-1B and Google Gemini 2.0 Flash for every experiment described to address all our research question.
- There was one instance where we encountered an issue with llama, where it was generating the same input prompt in the output. We took assistance from chatgpt to debug the issue. Apparently, it was happening because the input prompt was too large that it overflowed the input token context for LLM. Model settings was given in the prompt and gpt was asked to modify it to resolve the issue.
- We also used chatgpt for generating basic structure of code for visualization pipeline designed for knowledge graphs and for prompting code. We gave the details of graph and dataset structure in the prompt.

- We used LLM's to refine our sentence structure and correct any grammatical errors in our report. We do not use LLM's to generate any content but as a guiding hand in refining the content.

## A.2 Prompts Used for Experiments

- 1. Vanilla Prompting Meta Llama 3.2-1B  Gemini 2.0 Flash

```
---------------
Task: Optimize the following Python code to improve efficiency and make it
more concise.

Do not explain or use comments, only return the optimized code. Give a code
according to Python 3.8 and the whole answer should be enclosed
in a Class Solution and function name should be the same as submitted to
you in the input code

Input:
{inefficient-code}

Output:
---------------
```

- 2. Prompt to generate reasoning from Gemini 2.0 Flash

```
---------------
Analyze the following code snippet. Return the following in a JSON format:
1. Its thought process (short) as to if this is an inefficient code or not.
2. The reason behind it not being an efficient code.
3. The sentiment behind its answer (choose between "positive", "neutral",
"negative").
4. The confidence of its answer (choose between "Highly confident", "Average
confidence", "Low confidence").

Code:
{code}
---------------
```

- 3. Self-feedback loop prompt to generate runtime and memory efficient code

```
---------------
### Task: Optimize the following Python code to improve
efficiency and make it more concise.

Do not explain or use comments, only return the optimized code.
Give a code according to Python 3.8 and the whole answer
should be enclosed in a Class Solution and function name
should be the same as submitted to you in the input code.
#### Input Code: {input code}
#### Optimized Code: {output}
```

```
---------------
```

- 4. Self-feedback loop prompt to generate Natural Language feedback code after we got execution results from LLM generated code from above prompt:

```
---------------
###Task : Give feedback in english in atmost 20-30 words
for why the code solution below is incorrect or having a
runtime error or inefficient and how the program can be fixed.

Don’t generate the corrected code just tell the reasoning.
## Candidate solution:{input code}
##Result when executed:{status of the LLM generated code -
Successfully ran/Runtime Error/ Wrong Answer}
##Runtime:{runtime}
## Feedback for incorrectness/inefficiency and
how it can be improved: {output feedback}
---------------
```

- 5. Self-feedback loop prompt to generate refined code from the generated feedback and execution results:

```
---------------
### Task: Optimize the following Python code to improve
efficiency considering the feedback and execution time
and make it more concise.

Do not explain or use comments, only return the optimized
code.
Give a code according to Python 3.8 and the whole
answer
should be enclosed in a Class Solution and function name
should be the same as submitted to you in the input code
#### Input Code: {input code}
#### Feedback: {feedback}
#### Execution time: {runtime}
#### Optimized Code: {output code}

---------------
```

| **Topic** | **Vanilla prompting(%)** | **Reasoning-based(%)** | **Self-feedback(%)** |
|---|---|---|---|
| Array | 96.43 | 74 | 100 |
| Backtracking | 0 | 100 | 100 |
| Bit-manipulation | 0 | 100 | 100 |
| Depth-first-search | 100 | 100 | 100 |
| Divide-and-conquer | 0 | 0 | 0 |
| Dynamic-programming | 100 | 100 | 100 |
| Hash-table | 100 | 68.75 | 100 |
| Linked-list | 0 | 100 | 100 |
| Math | 92.31 | 60 | 95.24 |
| Stack | 0 | 0 | 0 |
| String | 100 | 75 | 100 |
| Tree | 100 | 100 | 100 |
| Trie | 0 | 100 | 100 |
| Two-pointers | 100 | 71.43 | 100 |

Table 5: Accepted Runtime Faster Solutions Compared To Input Code for **Meta Llama3.2-1B**

| **Topic** | **Vanilla prompting(%)** | **Reasoning-based(%)** | **Self-feedback(%)** |
|---|---|---|---|
| Array | 40.00 | 48.97 | 90.24 |
| Backtracking | 50.00 | 0.00 | 0.00 |
| Bit-manipulation | 0 | 33.33 | 50.00 |
| Depth-first-search | 0.00 | 100.00 | 100.00 |
| Divide-and-conquer | 100.00 | 100.00 | 0 |
| Dynamic-programming | 0 | 0.00 | 50.00 |
| Hash-table | 37.50 | 41.17 | 55.55 |
| Linked-list | 0 | 25.55 | 100 |
| Math | 0 | 60 | 95.24 |
| Stack | 75 | 0 | 0 |
| String | 100 | 76.47 | 100 |
| Tree | 100 | 50 | 100 |
| Trie | 0 | 100 | 100 |
| Two-pointers | 71.45 | 85.71 | 100 |

Table 6: Accepted Runtime Faster Solutions Compared To Input Code for Google Gemini 2.0 Flash

| **Prompting Type** | **Maintainability Index Better (%)** | **Cyclomatic Complexity Better (%)** | **Halstead Volume Better (%)** |
|---|---|---|---|
| **LLAMA Vanilla Prompting** | 16.88 | 0.00 | 5.19 |
| **GEMINI Vanilla Prompting** | 20.90 | 0.75 | 19.40 |
| **LLAMA Reasoning Prompting** | 15.13 | 0.00 | 0.00 |
| **GEMINI Reasoning Prompting** | 20.00 | 0.83 | 15.83 |
| **LLAMA Self Feedback Prompting** | 65.33 | 54.67 | 61.33 |
| **GEMINI Self Feedback Prompting** | 49.04 | 45.19 | 50.96 |

Table 7: LLM Code Quality Comparison for Different Prompting Techniques