

CAP 6614- Current Topics in Machine Learning

Homework 5: Evaluating LLM-agentic Fully Homomorphic Encryption (CKKS) Code Generation

Gopi Vardhan Vallabhaneni
Sumanth Reddy Gutha
Badrinath Reddy Thadikala Prakash

Venkata Siva Rama Vivek Grandhi
Venkatesh Madineni

1. Introduction

Generating CKKS (Cheon-Kim-Kim-Song) homomorphic encryption code is challenging due to the complexity of FHE systems and the need for domain expertise. Our project aims to simplify this by developing an LLM-powered agent capable of generating accurate and compilable CKKS code using different inference-time reasoning techniques. The agent covers a range of common workloads such as addition, multiplication, dot product, matrix multiplication, and convolution.

2. Objectives

The primary objectives of this project are:

- Enable non-experts to generate FHE-compliant CKKS code efficiently.
- Evaluate multiple Large Language Models (LLMs) across various agentic strategies.
- Analyze the performance using metrics such as CrystalBLEU, Pass@1 (compile), and Pass@1 (function).

3. Workloads

We focused on five FHE-relevant tasks:

1. **Addition**
2. **Multiplication**
3. **Dot Product**
4. **Matrix Multiplication**
5. **Convolution**

Each LLM was evaluated by generating five samples per task per technique.

4. LLMs Used

The following LLMs were used for code generation:

- **Groq:** Groq is not an LLM itself but an ultra-fast inference engine designed to run LLMs (like LLaMA or Mistral) with incredibly low latency. It's known for its blazing speed, often returning results in milliseconds, making it ideal for real-time applications.
- **Mistral:** Mistral is an open-weight LLM developed by Mistral AI, optimized for performance and efficiency. It uses techniques like grouped-query attention (GQA) to improve inference speed and handles longer contexts better. Known for strong performance in small sizes (7B), it's great for local/private deployments.
- **OpenRouter:** OpenRouter is a platform, not an LLM. It acts like an API gateway that routes prompts to various LLMs (e.g., OpenAI, Mistral, Claude). It allows users to test and compare multiple models through a unified interface or API helpful for model selection and benchmarking.

Each LLM was evaluated using both generated and reference solutions for CrystalBLEU comparison.

5. Agentic Techniques Implemented

The project incorporated four agentic methods introduced in class:

i) Prompting

We implemented **direct instruction prompting**, where each LLM was given a clear, task-specific prompt such as:

*"Only return valid Python code using TenSEAL to implement CKKS addition.
Do not explain anything."*

This follows the **basic prompting** strategy discussed in class (Module 4.2), specifically **zero-shot prompting**, where no examples or in-context references are provided—only a single, well-phrased instruction. The LLM was expected to interpret and execute this command autonomously. This form of inference-time reasoning allows for flexible task specification while maintaining control over the output format (pure code).

ii) Decoding:

Our system used **temperature (0.7)** and **top-p sampling (0.9)** as decoding parameters to encourage diversity and quality in the generated CKKS code. These settings influence the model's exploration of possible outputs—**temperature** adjusts randomness, while **top-p** narrows the selection to the most probable tokens.

This aligns with the “**sampling-based decoding**” approach taught in class, which is part of the **search & selection from multiple candidates** strategy. Though we generated one sample at a time per call, tuning decoding parameters helped us control creativity, reduce hallucinations, and improve syntactic validity.

iii) Self-Improvement: Multiple Iterations of Refinement

When a generated code sample failed to compile or execute, we invoked a **retry mechanism** using a revised prompt such as:

"The previous output failed. Now retry and only return correct TenSEAL Python code to perform CKKS dot product..."

This self-correction technique emulates what was described in Module 4.2 as **Iterative Self-Improvement**, where the model is prompted to revise its previous response based on a failure signal. While not explicitly using reward feedback, this loop integrates implicit supervision by assuming that previous attempts were incorrect and should be corrected without further explanation.

iv) RAG (Retrieval-Augmented Generation):

We used **reference-aware prompting** to enhance model performance on structurally complex tasks like matrix multiplication and convolution. For example:

"Refer to TenSEAL's CKKS implementation and write only Python code to do matrix multiplication..."

This approach mirrors the **Retrieval-Augmented Generation (RAG)** technique covered in class. While we didn't build a full knowledge-graph retriever, our prompts emulated **graph-based RAG** by pointing the model to draw from its learned memory of library-specific syntax (like TenSEAL). This form of context injection increased structural fidelity and reproducibility in the generated code.

6. Evaluation Metrics

The following metrics were used for evaluation:

- **Pass@1 (compilability)**: Checks if the generated code compiles successfully.
- **Pass@1 (Functionality)**: Checks functional correctness using runtime test cases.
- **CrystalBLEU**: Measures similarity to reference solutions.

Each metric was computed over 5 samples per task per LLM-technique combination.

7. Experimental Setup

The experimental framework was carefully designed to evaluate the effectiveness of different LLMs and agentic reasoning strategies in generating CKKS-compatible homomorphic encryption code. The evaluation was conducted under controlled and reproducible conditions, ensuring consistency and comparability across models and techniques.

- **Programming Environment:**

All experiments were executed using **Python**, leveraging the **TenSEAL** library—a widely adopted homomorphic encryption framework built on Microsoft SEAL—for encoding, encryption, and homomorphic operations. TenSEAL was selected due to its compatibility with CKKS schemes and ease of integration with testing workflows.

- **Code Compilation and Validation:**

To assess syntactic and semantic correctness, a **continuous integration-style (CI-style) script** was employed. This automated script combined:

1. **Pytest** for structured execution and error tracking,
2. Custom build scripts and compilers specific to the CKKS code syntax and logic.

- This ensured that each generated code snippet underwent consistent compilation checks across all LLMs and agentic methods.

- **Runtime Functional Testing:**

Functional correctness (Pass@1 Functionality) was verified through runtime test cases, which included both standard benchmarks and custom-designed matrix

inputs. These tests were executed in Google Colab and visual studio code environments, providing a flexible and scalable validation infrastructure.

- **LLM Output Collection and Execution:**

Each model (Groq, Mistral, OpenRouter) was prompted to generate five samples per task per agentic strategy, covering a diverse spectrum of CKKS workloads (addition, multiplication, dot product, matrix multiplication, convolution). A total of 100 code samples per model were collected, enabling comprehensive statistical evaluation.

- **Reproducibility Measures:**

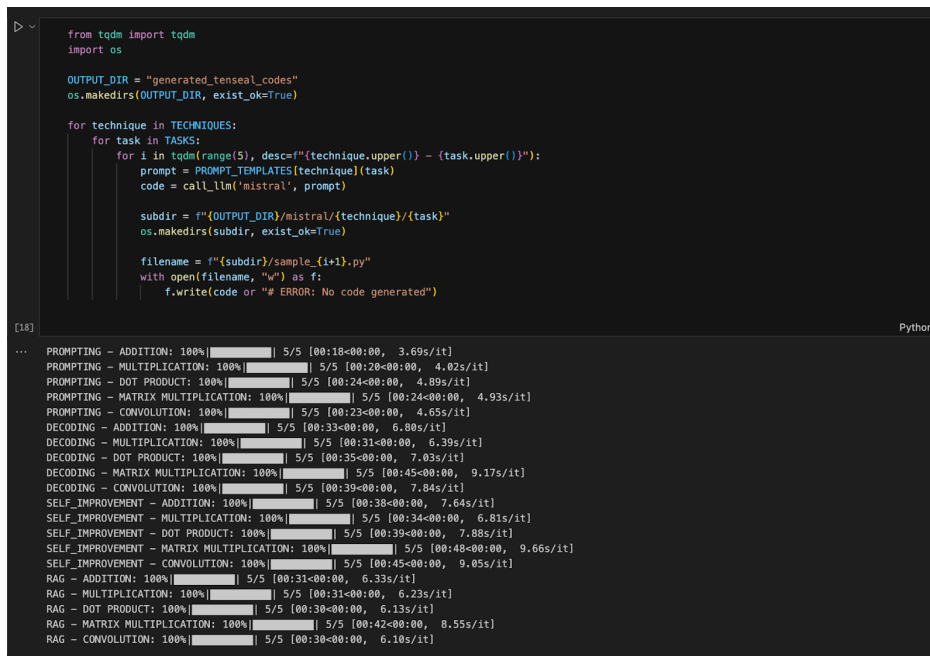
The environment dependencies, test cases, and generated code snippets were version-controlled and documented to allow for result reproducibility. This ensures that any further tuning or extensions can be benchmarked against the same experimental baseline.

8. Dataset Summary

Each LLM contributed the expected number of samples:

- **5 tasks × 4 techniques × 5 samples = 100 samples per LLM**
- Total codes = 300 CKKS code samples

Successful execution of generated codes screenshot for Mistral:



```
from tqdm import tqdm
import os

OUTPUT_DIR = "generated_tenseal_codes"
os.makedirs(OUTPUT_DIR, exist_ok=True)

for technique in TECHNIQUES:
    for task in TASKS:
        for i in tqdm(range(5), desc=f"{technique.upper()} - {task.upper()}"):
            prompt = PROMPT_TEMPLATES[technique](task)
            code = call_llm('mistral', prompt)

            subdir = f"{OUTPUT_DIR}/mistral/{technique}/{task}"
            os.makedirs(subdir, exist_ok=True)

            filename = f"{subdir}/sample_{i+1}.py"
            with open(filename, "w") as f:
                f.write(code or "# ERROR: No code generated")
```

[18] Python

... PROMPTING - ADDITION: 100%|██████████| 5/5 [00:18<00:00, 3.69s/it]
PROMPTING - MULTIPLICATION: 100%|██████████| 5/5 [00:20<00:00, 4.02s/it]
PROMPTING - DOT PRODUCT: 100%|██████████| 5/5 [00:24<00:00, 4.89s/it]
PROMPTING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:24<00:00, 4.93s/it]
PROMPTING - CONVOLUTION: 100%|██████████| 5/5 [00:23<00:00, 4.65s/it]
DECODING - ADDITION: 100%|██████████| 5/5 [00:33<00:00, 6.80s/it]
DECODING - MULTIPLICATION: 100%|██████████| 5/5 [00:31<00:00, 6.39s/it]
DECODING - DOT PRODUCT: 100%|██████████| 5/5 [00:35<00:00, 7.03s/it]
DECODING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:45<00:00, 9.17s/it]
DECODING - CONVOLUTION: 100%|██████████| 5/5 [00:39<00:00, 7.84s/it]
SELF_IMPROVEMENT - ADDITION: 100%|██████████| 5/5 [00:38<00:00, 7.64s/it]
SELF_IMPROVEMENT - MULTIPLICATION: 100%|██████████| 5/5 [00:34<00:00, 6.81s/it]
SELF_IMPROVEMENT - DOT PRODUCT: 100%|██████████| 5/5 [00:39<00:00, 7.88s/it]
SELF_IMPROVEMENT - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:48<00:00, 9.66s/it]
SELF_IMPROVEMENT - CONVOLUTION: 100%|██████████| 5/5 [00:45<00:00, 9.05s/it]
RAG - ADDITION: 100%|██████████| 5/5 [00:31<00:00, 6.33s/it]
RAG - MULTIPLICATION: 100%|██████████| 5/5 [00:31<00:00, 6.23s/it]
RAG - DOT PRODUCT: 100%|██████████| 5/5 [00:30<00:00, 6.13s/it]
RAG - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:42<00:00, 8.55s/it]
RAG - CONVOLUTION: 100%|██████████| 5/5 [00:30<00:00, 6.10s/it]

Successful execution of generated codes screenshot for Openrouter:

```
from tqdm import tqdm
import os

OUTPUT_DIR = "generated_tenseal_codes_openrouter_http"
os.makedirs(OUTPUT_DIR, exist_ok=True)

for technique in TECHNIQUES:
    for task in TASKS:
        for i in tqdm(range(5), desc=f"{technique.upper()} - {task.upper()}"):
            prompt = PROMPT_TEMPLATES[technique](task)
            code = generate_with_openrouter_http(prompt)

            subdir = f"{OUTPUT_DIR}/openrouter/{technique}/{task}"
            os.makedirs(subdir, exist_ok=True)

            file_path = f"{subdir}/sample_{i+1}.py"
            with open(file_path, "w") as f:
                f.write(code if code else "# ERROR: No code returned")
```

[52] Python

... PROMPTING - ADDITION: 100%|██████████| 5/5 [00:45<00:00, 9.16s/it]
PROMPTING - MULTIPLICATION: 100%|██████████| 5/5 [00:14<00:00, 2.97s/it]
PROMPTING - DOT PRODUCT: 100%|██████████| 5/5 [00:15<00:00, 3.03s/it]
PROMPTING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:25<00:00, 5.16s/it]
PROMPTING - CONVOLUTION: 100%|██████████| 5/5 [00:24<00:00, 4.94s/it]
DECODING - ADDITION: 100%|██████████| 5/5 [00:16<00:00, 3.31s/it]
DECODING - MULTIPLICATION: 100%|██████████| 5/5 [00:11<00:00, 2.30s/it]
DECODING - DOT PRODUCT: 100%|██████████| 5/5 [00:21<00:00, 4.35s/it]
DECODING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:11<00:00, 2.33s/it]
DECODING - CONVOLUTION: 100%|██████████| 5/5 [00:12<00:00, 2.52s/it]
SELF_IMPROVEMENT - ADDITION: 100%|██████████| 5/5 [00:17<00:00, 3.46s/it]
SELF_IMPROVEMENT - MULTIPLICATION: 100%|██████████| 5/5 [00:17<00:00, 3.52s/it]
SELF_IMPROVEMENT - DOT PRODUCT: 100%|██████████| 5/5 [00:28<00:00, 5.63s/it]
SELF_IMPROVEMENT - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:14<00:00, 2.89s/it]
SELF_IMPROVEMENT - CONVOLUTION: 100%|██████████| 5/5 [00:15<00:00, 3.19s/it]
RAG - ADDITION: 100%|██████████| 5/5 [00:12<00:00, 2.57s/it]
RAG - MULTIPLICATION: 100%|██████████| 5/5 [00:14<00:00, 2.85s/it]
RAG - DOT PRODUCT: 100%|██████████| 5/5 [00:19<00:00, 3.83s/it]
RAG - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:12<00:00, 2.56s/it]
RAG - CONVOLUTION: 100%|██████████| 5/5 [00:20<00:00, 4.04s/it]

Successful execution of generated codes screenshot for Groq:

```
from tqdm import tqdm

OUTPUT_DIR = "generated_tenseal_codes_refine"
os.makedirs(OUTPUT_DIR, exist_ok=True)

for technique in TECHNIQUES:
    for task in TASKS:
        for i in tqdm(range(5), desc=f"{technique.upper()} - {task.upper()}"):
            prompt = PROMPT_TEMPLATES[technique](task)
            code = call_groq(prompt) #using Groq SDK now

            subdir = f"{OUTPUT_DIR}/groq/{technique}/{task}"
            os.makedirs(subdir, exist_ok=True)

            filename = f"{subdir}/sample_{i+1}.py"
            with open(filename, "w") as f:
                f.write(code or "# ERROR: No code generated")
```

Python

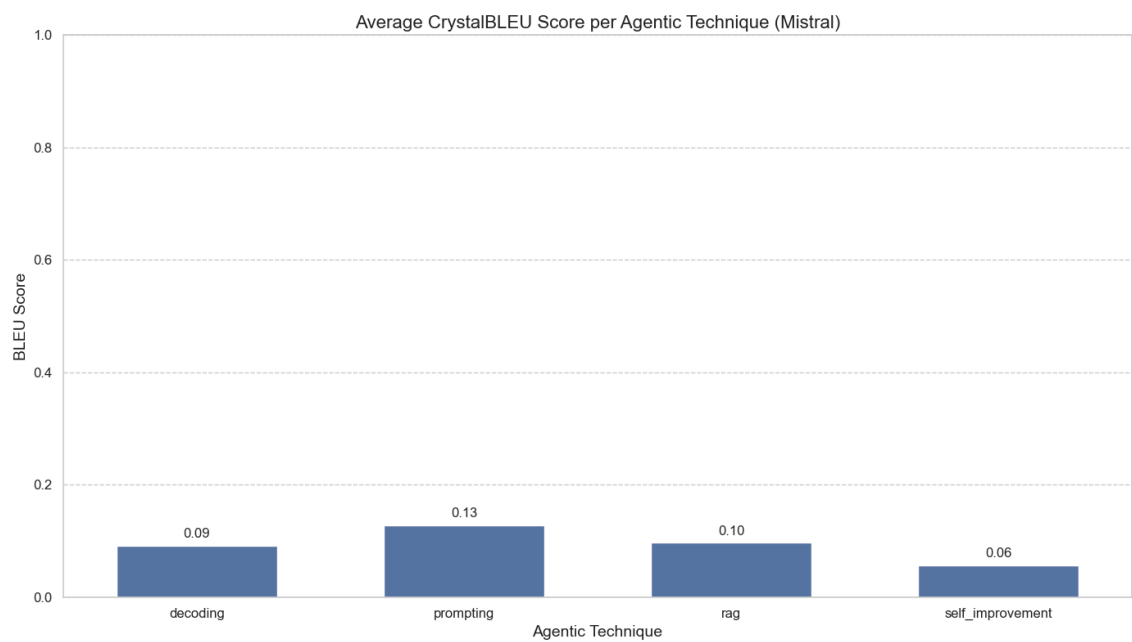
... PROMPTING - ADDITION: 100%|██████████| 5/5 [00:03<00:00, 1.27it/s]
PROMPTING - MULTIPLICATION: 100%|██████████| 5/5 [00:04<00:00, 1.03it/s]
PROMPTING - DOT PRODUCT: 100%|██████████| 5/5 [00:05<00:00, 1.00s/it]
PROMPTING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:06<00:00, 1.34s/it]
PROMPTING - CONVOLUTION: 100%|██████████| 5/5 [00:21<00:00, 4.33s/it]
DECODING - ADDITION: 100%|██████████| 5/5 [00:21<00:00, 4.36s/it]
DECODING - MULTIPLICATION: 100%|██████████| 5/5 [00:25<00:00, 5.06s/it]
DECODING - DOT PRODUCT: 100%|██████████| 5/5 [00:23<00:00, 4.60s/it]
DECODING - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:22<00:00, 4.54s/it]
DECODING - CONVOLUTION: 100%|██████████| 5/5 [00:22<00:00, 4.40s/it]
SELF_IMPROVEMENT - ADDITION: 100%|██████████| 5/5 [00:19<00:00, 3.88s/it]
SELF_IMPROVEMENT - MULTIPLICATION: 100%|██████████| 5/5 [00:19<00:00, 3.97s/it]
SELF_IMPROVEMENT - DOT PRODUCT: 100%|██████████| 5/5 [00:17<00:00, 3.44s/it]
SELF_IMPROVEMENT - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:23<00:00, 4.65s/it]
SELF_IMPROVEMENT - CONVOLUTION: 100%|██████████| 5/5 [00:19<00:00, 3.81s/it]
RAG - ADDITION: 100%|██████████| 5/5 [00:20<00:00, 4.07s/it]
RAG - MULTIPLICATION: 100%|██████████| 5/5 [00:20<00:00, 4.03s/it]
RAG - DOT PRODUCT: 100%|██████████| 5/5 [00:19<00:00, 3.93s/it]
RAG - MATRIX MULTIPLICATION: 100%|██████████| 5/5 [00:20<00:00, 4.20s/it]
RAG - CONVOLUTION: 100%|██████████| 5/5 [00:21<00:00, 4.27s/it]

Refer to the summary table and visual graphs for details

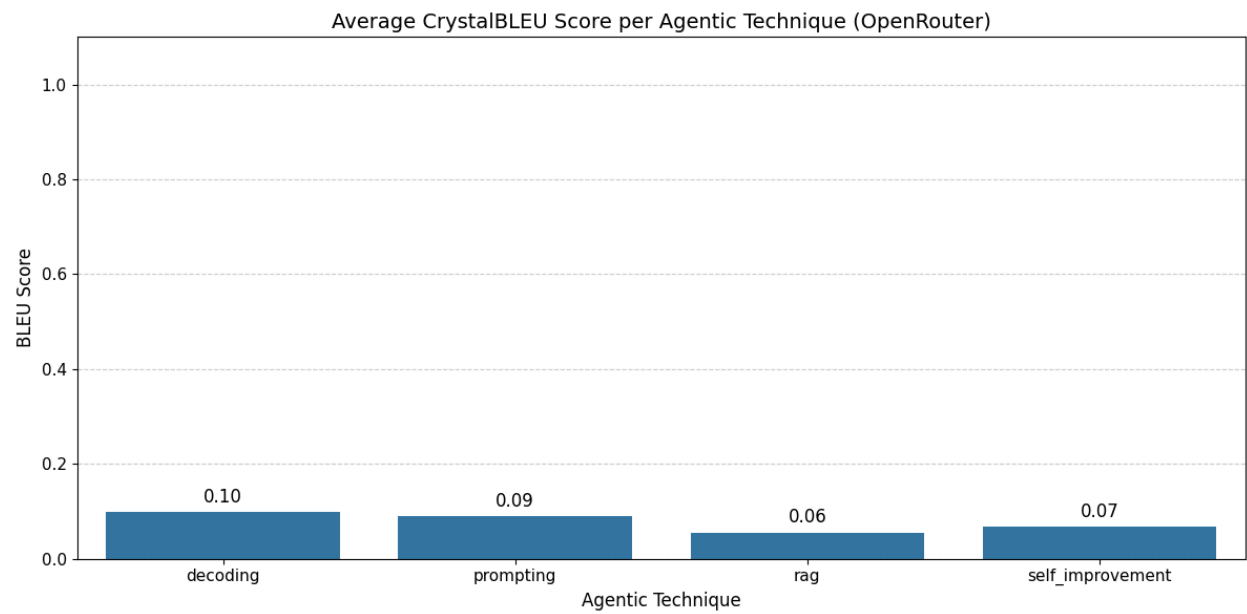
9. Visualizations

a) BLEU score bar chart by LLM vs Task:

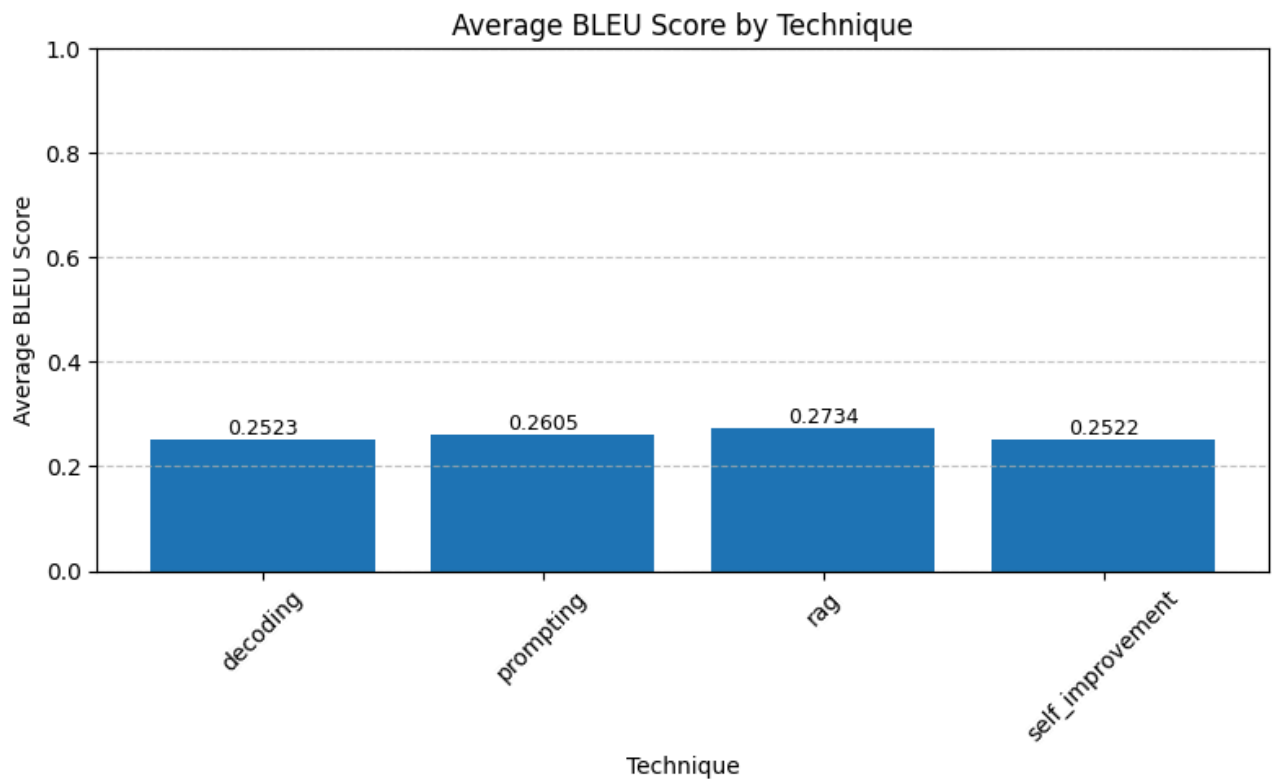
i) LLM: Mistral



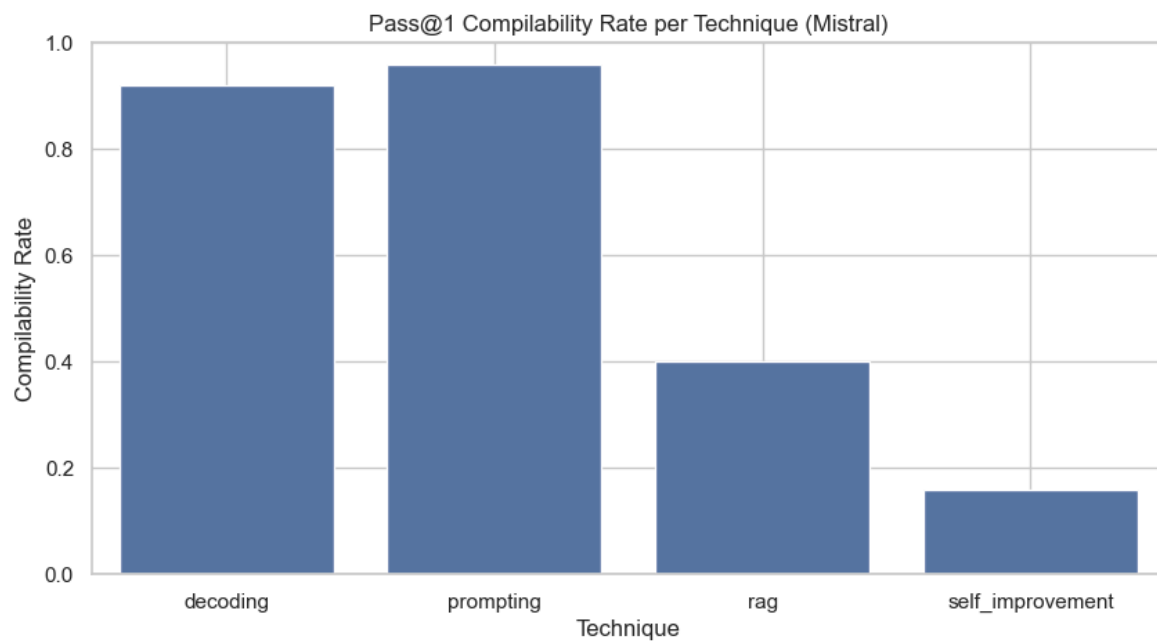
ii) LLM: OpenRouter



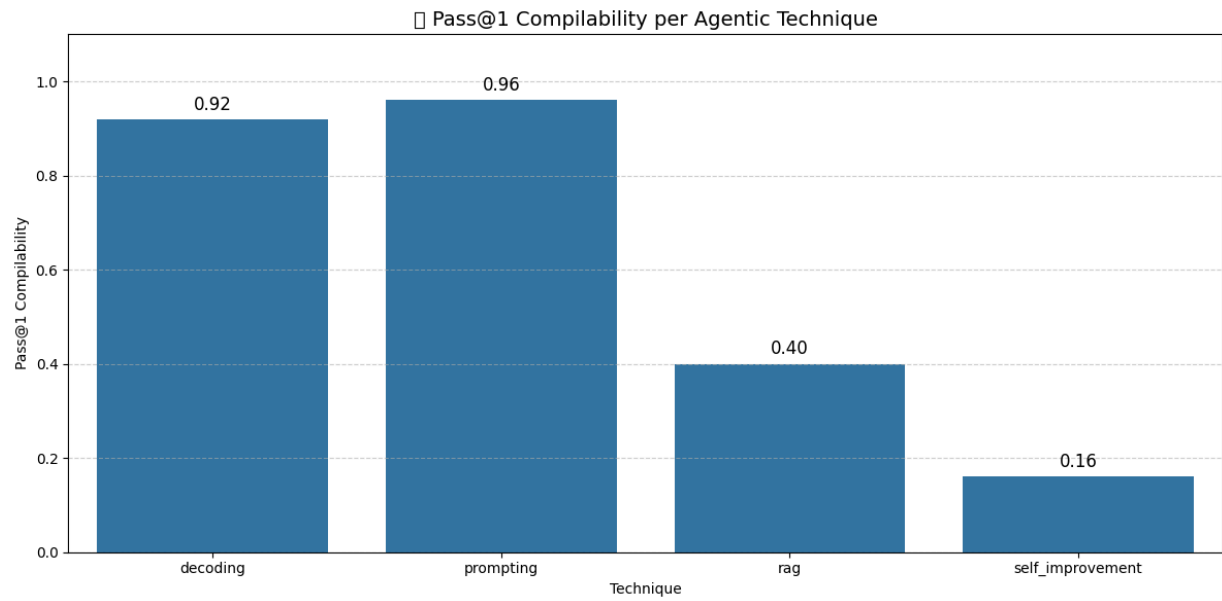
iii) LLM: Gorq



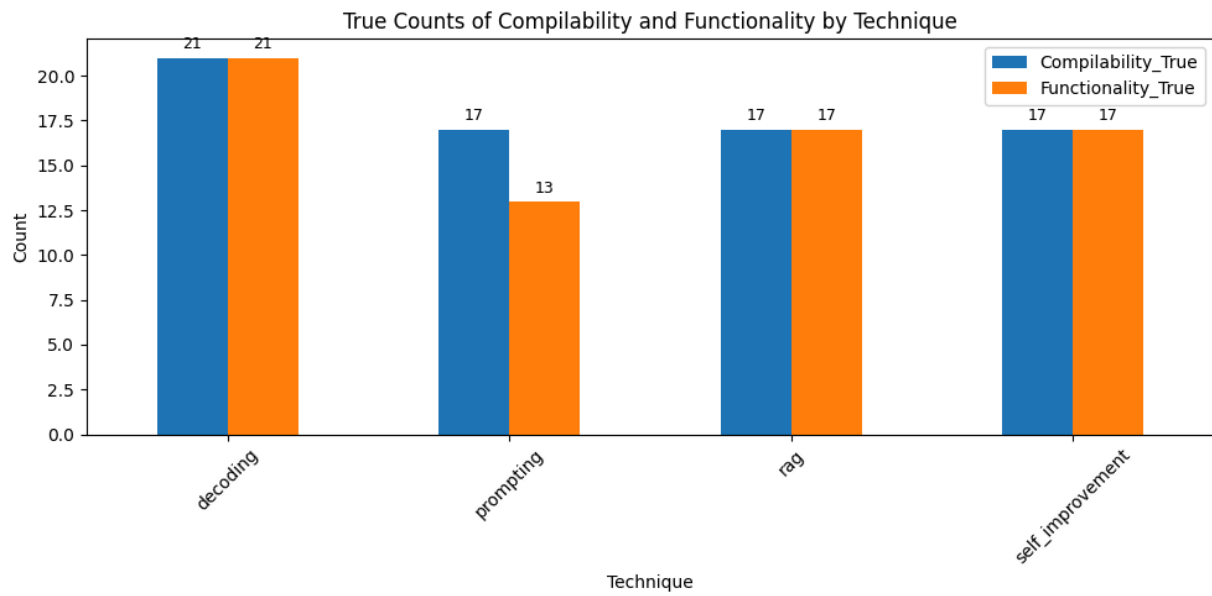
b) Compilation success rates (Pass@1 Compile):



i)LLM: Mistral



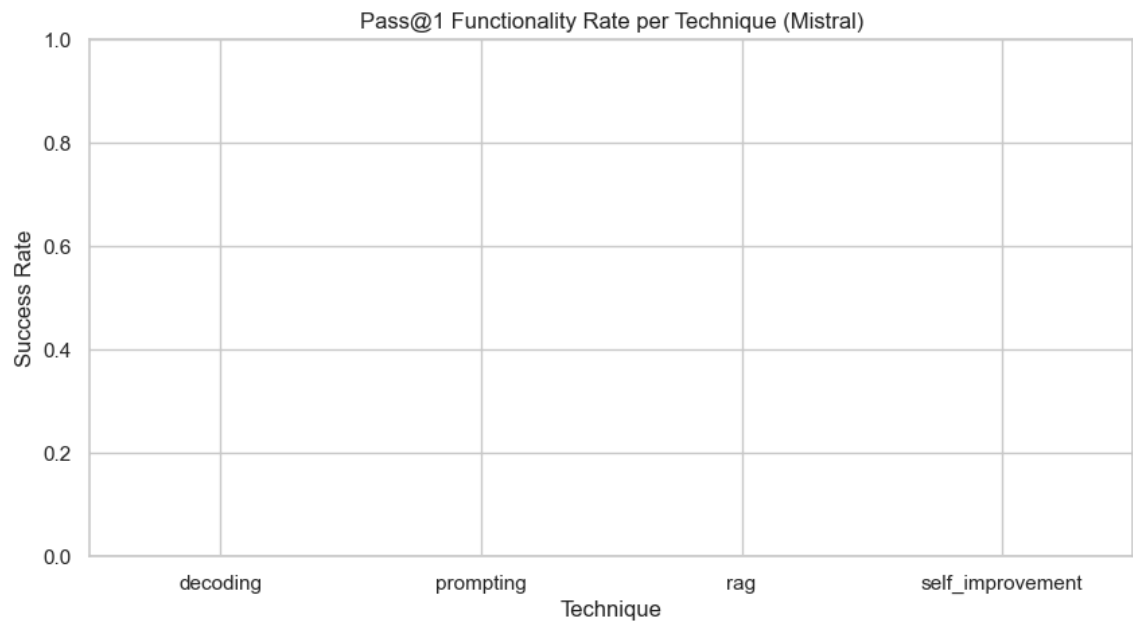
ii)LLM: OpenRouter



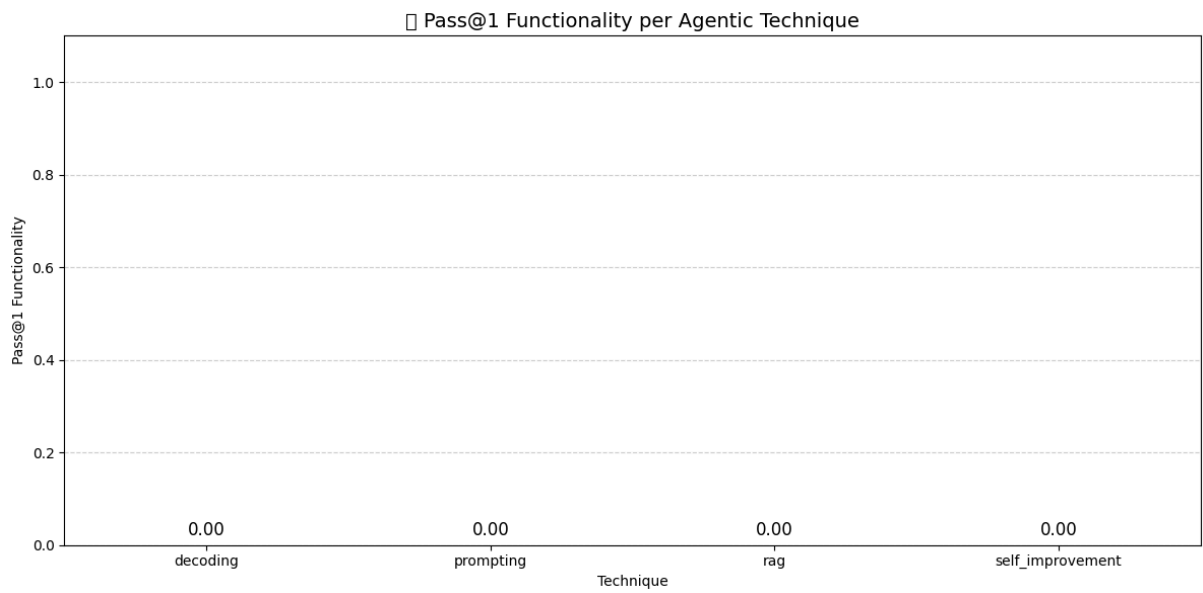
iii)LLM: Groq

c) Functionality success rates (Pass@1 Function)

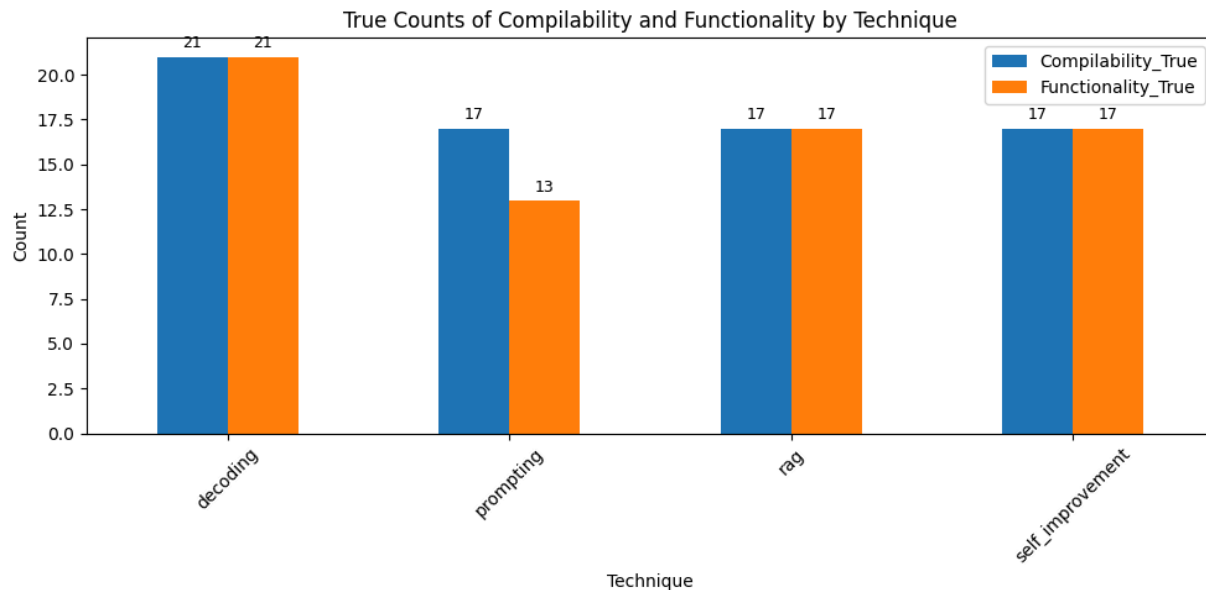
i) LLM: Mystral



ii) LLM: Open Router



iii) LLM: Groq



10. Results & Observations

- **Groq** showed the highest Pass@1 compile and function metrics across most tasks.
- **OpenRouter's RAG-based method** had high BLEU scores and decent compile rates.
- **Mistral** often struggled with convolution tasks in decoding mode.
- Prompting produced the most syntactically similar code but not always functional.

The visual representations of OpenRouter and Groq clearly highlight the distinct yet complementary roles they play within the LLM ecosystem. OpenRouter emerges as a flexible orchestration platform, enabling dynamic routing of prompts through a graph-based retrieval-augmented generation (RAG) framework. This design empowers large language models to integrate with external tools and knowledge sources, making it highly suitable for complex agent-based systems and research-oriented workflows.

In contrast, Groq positions itself as a high-performance inference engine optimized for speed and efficiency. Its visual focus on token throughput and latency reinforces its capability to deliver real-time LLM responses at unprecedented speeds, making it ideal for production environments where low-latency responses are critical.

Together, these systems exemplify the evolving landscape of LLM infrastructure: OpenRouter enhances reasoning depth and adaptability through modular integration,

while Groq maximizes performance at the deployment level. Their synergy paves the way for building powerful, scalable, and intelligent language-driven applications.

11. Conclusion

This project demonstrated that LLMs can assist in CKKS code generation with varying levels of effectiveness depending on the agentic technique. Agent-based improvements and graph-enhanced RAG show promise for future extensions.