POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

RP7.1: LLMs for Code Generation

Exam of the course:
054443 - SOFTWARE ENGINEERING 2

Tito Nicola Drugman: 10847631, Giacomo Colosio: 10775899, Patrizio Acquadro: 11087897

Lecturers:
Matteo Camilli,
Davide Yi Xian Hu,
Vincenzo Scotti,
Giovanni Ennio Quattrocchi

Academic year:
2025-2026

Abstract: This project investigates whether retrieval-augmented generation (RAG) and prompt engineering improve large language models (LLMs) for library-centric code generation. Using CodeLlama-7B-Instruct, we evaluate seven prompt families (from minimal task-to-code to checklist-driven and formal specification styles) under baseline (no retrieval) and RAG settings. The knowledge base contains more than 7.6k code snippets from two target libraries (PySCF and SEED-Emulator); evaluation uses 23 Long Code Arena tasks filtered to those libraries. We compare three single-hop retrievers—BM25 (lexical), CodeBERT cosine (semantic), and a hybrid via Reciprocal Rank Fusion—and two multi-hop pipelines (decomposition and iterative refinement). Results show that semantic retrieval with CodeBERT consistently and substantially outperforms both the baseline and BM25, with the best score of 0.2567 (prompt v2). The hybrid fusion helps over pure BM25 but trails pure semantic retrieval. Among multi-hop strategies, decomposition paired with a checklist-driven prompt (v8) reaches 0.2505, rivaling the best single-hop semantic setup; iterative multi-hop achieves comparable performance on several prompts. Minimal prompts work best for the baseline (0.1859 with v1), suggesting that added structure without external context can introduce overhead.

Key-words: LLMs; code generation; retrieval-augmented generation (RAG); CodeLlama-7B; BM25; CodeBERT; cosine similarity; multi-hop retrieval; prompt engineering; Long Code Arena (LCA).

1. Introduction

This report documents the project completed for the course *Software Engineering 2* of Professor Di Nitto Elisabetta, investigating large language models (LLMs) as code assistants. We study whether augmenting the LLM prompt with retrieved code snippets (alongside explicit instructions and the user request) improves the quality of the generated code compared to using the same prompt without snippets. We conducted our evaluations on the following workstation:

- (i) a node with 2× NVIDIA GeForce RTX 5060 Ti (16 GB each), 16 CPU cores, and 128 GB RAM.

.
In this report, we analyze the following dimensions:

- **Prompt design**: how alternative prompt formulations influence the quality of the code produced by the LLM.
- **Baseline vs. RAG**: whether a baseline LLM (given only instructions and the user request) performs better or worse than the same LLM augmented with retrieved code snippets (Retrieval-Augmented Generation, RAG).

- **Snippet retrieval**: which strategy is most effective for identifying the top-k relevant snippets from the knowledge base.
- **Multi-hop pipelines**: comparison between decomposition-based and iterative refinement strategies, highlighting their respective impact on code quality.

2. Specifications

Our experimental workflow, depicted in Figure 1, is structured into five distinct phases that constitute an end-to-end pipeline for RAG-enhanced code generation and evaluation. This architecture was designed to be comprehensive, scalable, and facilitate high-performance experimentation.

- Phase 1: Data Preparation is the foundational stage where we define our knowledge sources and evaluation tasks. It begins with the LCA Dataset, from which we filter 23 specific queries related to the seed-emulator and pyscf libraries. Concurrently, we establish our Knowledge Base made of the library repositories which are stored in a JSON format.
- Phase 2: Information Retrieval is the core of the RAG process, where we employ multiple strategies to find relevant code snippets. This phase is divided into two main branches: Standard Retrievers and the MultiHop Engine. The standard branch includes lexical search with BM25, semantic search using Cosine Similarity and a Hybrid method combining both via Reciprocal Rank Fusion (RRF). The advanced MultiHop Engine implements both a Decomposition strategy, which breaks the main query into sub-queries for parallel retrieval, and an Iterative strategy, which refines the search over multiple steps.
- Phase 3: Prompt Construction takes the snippets retrieved in the previous phase and systematically assembles them into prompts for the LLM. Using a flexible Template System, we generate the full prompts for both Baseline (no RAG) and RAG configurations across some distinct template variants.
- Phase 4: Code Generation is where the constructed prompts are passed to the CodeLlama-7b-Instruct model. To ensure deterministic and reliable outputs for comparison, we employ fixed generation parameters and a max tokens of 512. Our execution pipeline is designed to be robust, featuring resume-safe execution and out-of-memory (OOM) handling.
- Phase 5: The final stage of our pipeline is the evaluation of generated code. In our work, we adopt **CodeBLEU** [1] as the primary metric. CodeBLEU extends the traditional BLEU metric by incorporating additional dimensions that are specific to programming languages: an abstract syntax tree (AST) match, a data-flow match, and a weighted n-gram match. This makes it particularly suitable for evaluating code generation tasks, as it captures not only the surface-level similarity of tokens but also the syntactic and semantic structure of the programs. We compute CodeBLEU scores for each generated instance and aggregate the results across prompt variants and retrieval strategies. This enables a systematic comparative analysis between the *baseline*, the standard *RAG methods*, and the *multi-hop strategies*. By relying on CodeBLEU, we ensure that our evaluation reflects both lexical accuracy and structural correctness, providing a more reliable assessment than BLEU alone.

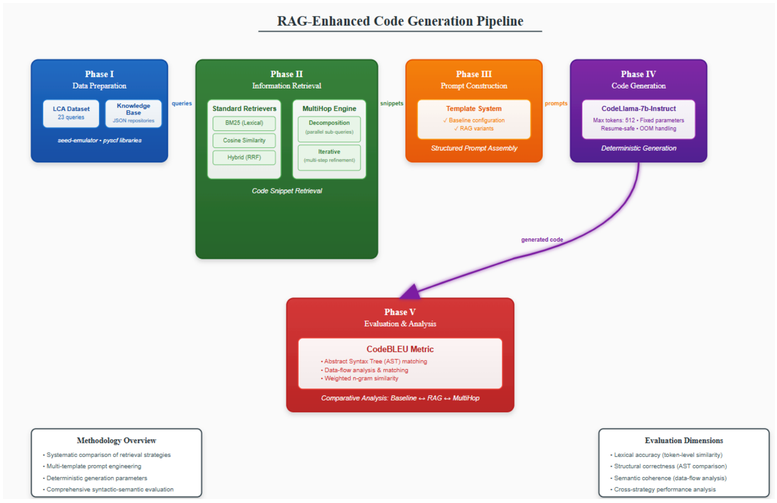
RAG-Enhanced Code Generation Pipeline
Phase I
Data Preparation
LCA Dataset
23 queries
Knowledge Base
JSON repositories
seed-emulator • pyscf libraries
queries
Phase II
Information Retrieval
Standard Retrievers
BM25 (Lexical)
Cosine Similarity
Hybrid (RRF)
MultiHop Engine
Decomposition
(parallel sub-queries)
Iterative
(multi-step refinement)
Code Snippet Retrieval
snippets
Phase III
Prompt Construction
Template System
✓ Baseline configuration
✓ RAG variants
Structured Prompt Assembly
prompts
Phase IV
Code Generation
CodeLlama-7b-Instruct
Max tokens: 512 • Fixed parameters
Resume-safe • OOM handling
Deterministic Generation
generated code
Phase V
Evaluation & Analysis
CodeBLEU Metric
• Abstract Syntax Tree (AST) matching
• Data-flow analysis & matching
• Weighted n-gram similarity
Comparative Analysis: Baseline ↔ RAG ↔ MultiHop
Methodology Overview
• Systematic comparison of retrieval strategies
• Multi-template prompt engineering
• Deterministic generation parameters
• Comprehensive syntactic-semantic evaluation
Evaluation Dimensions
• Lexical accuracy (token-level similarity)
• Structural correctness (AST comparison)
• Semantic coherence (data-flow analysis)
• Cross-strategy performance analysis

Figure 1: Architectural Overview of the RAG-Enhanced Code Generation Pipeline. The diagram illustrates the five distinct phases of our experimental setup, from initial data preparation to the final comprehensive evaluation of the generated code.

2.1. Large Language Models

We use `codellama/CodeLlama-7b-Instruct-hf`[1] (7B, instruction-tuned). We selected the 7B size to keep multihop and prompt-sweep experiments feasible on our hardware while retaining an instruction-following code model. The model's context window is 16,384 tokens.

Loading and quantization. We load the model with Hugging Face Transformers, preferring 4-bit quantization on CUDA and falling back to `bfloat16`/`float16` if 4-bit is unavailable. Quantization uses bitsandbytes NF4 with double quantization; the compute dtype is chosen dynamically: `bfloat16` if supported by the GPU, otherwise `float16`. After a successful load, the tokenizer and weights are cached locally together with a metadata JSON (model id + quant settings); on subsequent runs, if the metadata matches, we load entirely from cache. When the tokenizer lacks a padding token, we set `pad_token_id = eos_token_id` to ensure consistent decoding.
Concrete flags (for reproducibility):
- `load_in_4bit=True`
- `bnb_4bit_quant_type="nf4"`
- `bnb_4bit_use_double_quant=True`
- `bnb_4bit_compute_dtype` $\in$ {`bfloat16`, `float16`}

2.2. Prompts

We evaluate how prompt structure affects generated code. We define two prompt families:
- **Baseline prompts**, which present only the task to the LLM (no retrieved context).
- **RAG prompts**, which contains the top-k retrieved snippets to the task.

Full templates are provided in Appendix A; for completeness, we summarize the prompt templates here.

Version 1 — Minimal Task-to-Code. The baseline presents only the section headers and the task, while the RAG variant attach also the retrieved examples to the same structure. In both cases, the model is instructed to produce code after the *Code:* marker. The full prompt can be seen in Listing 1.

Version 2 – Senior Engineer This version adopts a persona-based approach, instructing the LLM to act as a senior Python engineer. The objective shifts from generating merely functional code to producing a complete, production-ready software artifact.
The prompts explicitly mandate a higher standard of quality by requiring:
- A well-structured function or class with type hints.
- Comprehensive documentation in the form of a docstring.
- Code verification through the inclusion of at least one unit test.

The RAG variant integrates this persona with retrieved examples, directing the "senior engineer" to use the provided snippets as a guide for their implementation. The full prompt can be seen in Listing 2.

Version 6.2 — Emphatic Style. A fully uppercase, block-structured prompt that demands a *single, runnable Python 3 file* solving the task, with heavy use of the target library inferred from the instruction. It enforces production-quality code (PEP 8), explicit imports, type hints, docstrings, and a brief usage example or simple test; the model must return *only* code, starting immediately after the `‘‘‘PYTHON` marker (no extra prose/markdown). The RAG variant prepends a `Retrieved Code Examples` section that supplies truncated snippets as context while preserving the same output constraints. The full prompt can be seen in Listing 3.

Version 6.3 – Explicit Quality and Correctness Mandate This version is a subtle but targeted refinement of the emphatic style introduced in V6.2. Its key distinction is the addition of an explicit and direct command for the generated code to be error-free ("THE SCRIPT MUST NOT GIVE ERRORS") and adhere to high-quality standards. This instruction, embedded within both the baseline and RAG variants, serves as an experiment in constraint reinforcement. It tests whether such a straightforward directive can measurably reduce the incidence of runtime errors and improve the overall robustness of the output, complementing the existing requirements for production-quality practices. The prompt otherwise retains the uppercase, commanding tone and formal structure of its predecessor. The full prompt can be seen in Listing 4.

[1] `https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf`

Version 7 – Professional Tone with Affective Reinforcement This version refines the structured approach by shifting from the emphatic, commanding tone of V6.2/V6.3 to a more professional and encouraging persona. While maintaining a highly organized format with clear sections for the task, requirements, and output, this prompt introduces two key changes. First, it places greater emphasis on high-level software engineering principles, explicitly asking for solutions that are "robust, modular, and easy to understand" and demonstrate "idiomatic use" of the target library. Second, it introduces a novel element of affective prompting, using a positive and motivational statement ("If you follow all the rules... you'll make us very proud") to encourage strict adherence to the specified constraints. This blend of professional expectations and positive reinforcement is designed to elicit high-quality, reliable code generation.

Version 8 – Checklist-Driven Generation with Meta-Instructions This version introduces a highly structured and modern prompt format, centered around an actionable checklist. It directly addresses the model as "You are ChatGPT, a world-class Python engineer," a meta-instructional persona. The core innovation is the replacement of prose requirements with an "Implementation Checklist" which guides the model through a clear sequence of quality gates covering planning, code quality, demonstration, and high-level software principles like robustness and "Clarity > cleverness." A sophisticated meta-instruction explicitly directs the model to plan its solution internally but suppress the plan from the final output, encouraging structured thought without verbose reasoning. The RAG variant maintains this clean structure, clearly delineating the problem from the retrieved examples and instructing the model to leverage and adapt patterns, thereby focusing its attention on synthesis.

Version 9 – Internal Workflow and API Grounding This version evolves the prompt by introducing an explicit, process-oriented "Workflow" for the model to follow internally. Instead of just a checklist of final requirements, this prompt outlines a cognitive sequence for the LLM: Analyse, Plan, and Self-review before writing any code. The core innovation lies within the "Self-review checklist," which includes a powerful new directive for the model to internally verify that every library call "exactly matches an existing API," targeting the common failure mode of API hallucination. The RAG variant reinforces this concept by framing the retrieved snippets as "verified code excerpts" and "grounding context," explicitly instructing the model to use this trusted information to ensure the correctness of its API usage and to reuse established patterns. This approach aims to instill a more disciplined and self-correcting generation process within the model.

2.3. Datasets and Knowledge Base

We rely on two distinct datasets: (i) a *knowledge base* (KB) from which to retrieve the top-k code snippets, and (ii) a *test set* used to evaluate model outputs.

2.3.1 Knowledge Base Dataset

For the KB, we use the *lca library-based code generation* dataset hosted on Hugging Face[2]. This resource contains several use cases of the Python libraries targeted by the benchmark tasks. Each of the 62 repository comprises many self-contained code snippets; we treat these repositories as the source of truth for the RAG component.
Due to time and complexity constraints, we restricted the KB to a subset of libraries: `pyscf` (Python-based Simulations of Chemistry Framework) and `SEED-Emulator`. Combined, these two repositories contributed more than 7,600 snippets, which we deem sufficient for this study. The snippets are intentionally heterogeneous in both length and complexity (e.g., ranging from short utility functions to full class implementations), which helps stress-test retrieval. Future work will extend coverage to additional libraries.

2.3.2 Test Dataset

For evaluation, we adopt the *lca library-based code generation* benchmark[3] curated by JetBrains Research. This dataset underpins the Long Code Arena (LCA), focusing on realistic, long-form software development tasks that require using specific third-party libraries. It contains 150 samples specifying the library repository owner (*repo_owner*), the natural-language task (*instruction*), and the corresponding reference implementation produced by the library authors (with and without comments: *reference*, *clean_reference*).

[2]`https://huggingface.co/datasets/JetBrains-Research/lca-library-based-code-generation/tree/main/repos`

[3]`https://huggingface.co/datasets/JetBrains-Research/lca-library-based-code-generation`

To ensure comparability with our KB, we filter the test set to tasks that target either `pyscf` or `SEED-Emulator`. This yields 23 evaluation items in total.

2.4. RAG Retrieval Augmented Generation

RAG is a technique that enables LLMs to retrieve and incorporate new information. By doing so the LLM can use domain-specific or updated information that was not available in the training data. RAG technique is able to improve LLMs by incorporating information retrieval before generating responses. Unlike traditional LLMs that rely on static training data, RAG pulls relevant text from databases, uploaded documents, or web sources. For our case RAG was used to retrieve code snippets of similar tasks to help the LLM achieve better results, the baseline did not receive any code snippet as input. There are several approaches on how to retrieve code snippets. We decided to evaluate three retrieval approaches.

2.4.1 BM25

One of the retrieval approaches we evaluated is the Okapi BM25 (Best Match 25), a state-of-the-art ranking function used by search engines. Unlike vector-based methods that rely on dense embeddings, BM25 is a probabilistic model based on the bag-of-words representation. It ranks a set of documents based on the query terms appearing in each document, without considering the semantic relationship between them. This makes it a powerful and efficient baseline for lexical relevance. The score is composed of three main components which are Inverse Document Frequency (IDF), Term Frequency (TF) and Document Length Normalization.

2.4.2 Cosine Similarity

To complement the lexical matching capabilities of BM25, we implemented a semantic retrieval approach based on cosine similarity. This method moves beyond keyword matching to capture the conceptual meaning of both the user's query and the code snippets in our knowledge base. By representing the user's prompt and all the snippets in a high-dimensional vector space, we can identify relevant snippets even when they do not share the exact same terminology as the query.
The foundation of our semantic retriever is the microsoft/codebert-base model which is pre-trained on a large corpus of source code. This choice ensures that the resulting embeddings are sensitive to the syntax and semantics of programming languages. The process begins by converting every snippet in our knowledge base into a 768-dimensional vector. For each snippet, the text is tokenized and fed into the CodeBERT model. Each post-processed vector is L2-normalized to have a unit length of 1.0 this is done since it makes the computationally efficient dot product between two vectors mathematically equivalent to their cosine similarity. The entire matrix of normalized snippet vectors is then cached on disk. When a user query is received, it undergoes the exact same transformation pipeline (except for caching). The resulting query vector is then multiplied with the pre-computed matrix of all snippet vectors to obtain the cosine similarity score between the query and every snippet in the knowledge base.

2.4.3 Hybrid Retriever

While the lexical (BM25) and semantic (Cosine Similarity) retrievers each possess distinct strengths, they also have inherent limitations. BM25 can fail when a query uses synonyms not present in the code, whereas the semantic model may overlook snippets that contain critical exact keywords. To create a more robust and balanced retrieval system, we implemented a hybrid approach that combines the outputs of both retrievers.
A significant challenge in fusing these two systems is that their raw scores are not directly comparable; BM25 scores are unbounded, while cosine similarity scores are confined to a [-1, 1] range. To circumvent this issue, we employ a rank-based fusion technique known as Reciprocal Rank Fusion (RRF) which combines multiple ranked lists without needing to normalize or interpret the underlying scores, making it ideal for our use case. RRF creates two independent rankings (one for BM25 and one for cosine similarity) and the hybrid RRF score for each snippet is then calculated by summing the reciprocal of its rank from each list, adjusted by a constant k which mitigate the influence of snippets that rank very highly in one system but poorly in the other.

2.5. MULTIHOP

While standard single-hop retrieval techniques are effective for straightforward queries, their performance often degrades when faced with complex, multi-step user instructions. A single search pass over a multifaceted query tends to perform a "semantic average", retrieving snippets that are only partially relevant to the overall goal or that address one sub-task while ignoring others. It is highly improbable that a single, perfect code snippet exists in the knowledge base that encapsulates all requirements of a complex prompt. To overcome this limitation, we investigated multi-hop retrieval strategies. These advanced methods leverage an auxiliary Large Language Model (LLM) as a reasoning engine to transform the single, complex retrieval problem into a more manageable series of targeted searches. The fundamental idea is to guide the search process dynamically, breaking down the problem to assemble a context that is both more comprehensive and precise than what a single-hop approach can provide. In our study, we implemented and evaluated two distinct multi-hop pipelines: a decomposition-based approach and an iterative refinement strategy.
We decided to test two different multihop approaches.

2.5.1 Multi-hop Decomposition: A Divide-and-Conquer Approach

The multi-hop decomposition strategy operates on a "divide and conquer" principle, designed to handle complex user queries that involve multiple distinct sub-tasks. The process begins by providing the user's full instruction to an auxiliary Large Language Model (LLM), which acts as a planner. Its role is to decompose the complex request into a set of smaller, more focused, and independent search sub-queries.
These sub-queries are then executed in parallel, with each triggering a separate single-hop retrieval pass against the knowledge base. The top-k snippets retrieved for each sub-query are then aggregated into a single, unified list. This final collection of snippets, which is inherently diverse and covers multiple facets of the original request, is then passed as context to the primary code-generating LLM and ensures that all components of a complex task are represented in the provided examples.

2.5.2 Multi-hop Iterative: An Exploratory Refinement Approach

In contrast to decomposition, the multi-hop iterative strategy functions as an exploratory search process that progressively refines its focus. The process commences with an initial retrieval pass (Hop 1) using the user's original, unmodified query. The top snippets from this first hop are then provided as context to an auxiliary LLM, alongside the original user request.
The LLM's objective is to analyze this context and generate a new, more technically precise search query for the next iteration (Hop 2). This refined query often incorporates specific function names, classes, or technical terms discovered in the snippets from the previous hop. This iterative cycle of retrieval and refinement continues for a predefined number of steps. The final set of snippets provided to the code-generating LLM is taken exclusively from the last, most focused iteration, ensuring the context is of the highest possible relevance. This method excels at homing in on the most pertinent information, even when the initial query is vague or lacks technical specificity.

3. Results

The result we obtained with using CodeLlama 7B are reported in Table 1.

The baseline relies solely on the LLM's internal knowledge without any retrieved snippets.

- Best Performance: The highest score is 0.185871 with prompt v1 which was the most minimal, suggesting that when the LLM has no external context to work with, a direct and uncluttered instruction is the most effective. More complex prompts likely add cognitive overhead without providing helpful information, leading the model to generate more boilerplate or misinterpret the core task.

.
This BM25 variant is good at finding snippets with exact term matches but lacks semantic understanding.

- Best Performance: The highest score is 0.168210 with prompt v8 which contains a highly structured "Implementation Checklist" that guides the LLM on planning, code quality, and robustness. Nevertheless the results obtained with only BM25 are in almost all the cases lower with respect to the baseline.

.
The cosine variant that uses CodeBERT embeddings is used to find semantically similar with code snippets.

- Best Performance: The highest score is 0.256719 with prompt v2, but all scores are consistently high and stable (around 0.25).

With respect to all the approaches we tested the cosine similarity is the most significant since it performs exceptionally well with respect to all the other tested variants across all prompt versions. This indicates that the semantic retriever is providing high-quality, relevant code examples. When the retrieved context is this good, the specific structure of the prompt becomes less critical. The LLM receives excellent guidance from the examples themselves, allowing it to succeed even with minimal prompting (v1) or highly structured ones (v8). This underscores a key principle that the quality of retrieval is more impactful than the complexity of the prompt engineering.

.

The hybrid variant combines the lexical ranks from BM25 and the semantic ranks from Cosine using Reciprocal Rank Fusion (RRF).

- Best Performance: The highest score is 0.215327 with prompt v8, RRF result in a mix of high and moderate-quality snippets. The v8 prompt helps the LLM synthesize this mixed-signal context effectively but the performance is in all the cases lower with respect to the cosine alone. The overall scores are better than pure BM25 but not as high or consistent as pure Cosine. This suggests that for this specific dataset, the semantic signal from CodeBERT is dominant, and the lexical signal from BM25 might be adding some noise that slightly degrades the top-tier performance of the pure semantic approach.

.

The multihop decomposition strategy first uses an LLM to break the main query into smaller, targeted sub-queries and then retrieves snippets for each.

- Best Performance: The highest score is 0.250543 with prompt v8 which is a top-tier performance higher than the best cosine similarity (which still was with prompt v8). The decomposition strategy is highly effective because it retrieves a diverse yet highly relevant set of snippets that cover different aspects of the main task. When these focused snippets are fed to the LLM, the structured guidance of the v8 checklist helps it synthesize them into a complete and correct final program. This shows that decomposition combined with advanced prompt engineering (checklist) yields better results that any other approach we evaluated.

.

Lastly the multihop iterative strategy first retrieves snippets, and then uses an LLM to generate a better query for the next "hop", refining the search over several steps.

- Best Performance: The highest score is 0.250354 with prompt v7. This method achieves with some prompts a performance comparable to the cosine one. The iterative refinement process can be able to retrieve the most relevant code snippets.

Table 1: Scores by prompt version and variant

variant	**v1**	**v2**	**v6_2**	**v6_3**	**v7**	**v8**	**v9**	Mean
baseline	0.185871	0.171419	0.141880	0.164756	0.144842	0.153567	0.172220	0.163410
bm25	0.161507	0.142972	0.096764	0.122803	0.133774	0.168210	0.087831	0.128609
cosine	0.253125	0.256719	0.250018	0.249077	0.250010	0.250002	0.250112	0.250732
hybrid	0.155605	0.154330	0.112124	0.129295	0.137169	0.215327	0.094614	0.138757
multihop_dec.	0.133053	0.182133	0.153410	0.206845	0.191376	0.250543	0.244997	0.200912
multihop_ite.	0.177532	0.213739	0.201527	0.223354	0.250354	0.250143	0.250154	0.224805

4. Conclusion

4.1. Conclusion

In this research we were able to run only a limited number of experiments. To accelerate iteration, we used the smallest CodeLlama model (7B). On our setup, each multihop execution required approximately 75–100 minutes.

A central challenge of this task is the inherently high variability of source code. The same functionality can be implemented in many equally correct ways—ranging from different variable names and stylistic choices to distinct control structures (e.g., *for* vs. *while*). Consequently, it is not always straightforward to determine whether two code snippets are functionally identical or whether one approach is objectively superior to another. The difficulty is also worsened by the diversity in how users phrase the same request, which can further influence both retrieval and generation.

4.2. Future Work

Several promising directions emerged but could not be explored due to time and complexity constraints. First, it would be valuable to refine prompting strategies. In particular, the instructions to the model could explicitly avoid forcing the use of retrieved snippets, since we cannot guarantee that the retrieved context is always optimal; assessing whether such changes improve performance is an open question.
Second, the knowledge base should be broadened. In this study it was limited to two libraries; expanding its coverage is likely to improve retrieval quality and, in turn, downstream generation.
Third, comparing different model families and sizes could clarify which models are better suited for code assistance. Running the same pipeline with larger variants—such as CodeLlama 34B or 70B—may yield better results, albeit at the cost of longer runtimes.
Finally, submitting the user's query to an auxiliary LLM to generate diverse paraphrases of the same prompt may improve robustness. Evaluating whether such paraphrasing enhances retrieval and multihop reasoning remains a compelling avenue for future investigation. But this would drastically impact the inference time since by generating 3 variations of the user's query we would need to execute the coe for a total of four times.

A. Appendix A

Here we show all the prompts we used.

A.1. Prompt 1

```python
def build_baseline_prompt_v1(instruction: str) -> str:
    return f"""\
### Task:
{instruction.strip()}

### Code:
"""

def build_rag_prompt_v1(instruction: str, retrieved: str) -> str:
    """
    RAG: prima gli esempi recuperati, poi il task, poi '### Code:'.
    """
    return f"""\
### Retrieved Examples:
{retrieved.strip()}

### Task:
{instruction.strip()}

### Code:
"""
```

Listing 1: Python code for building prompts v1.

A.2. Prompt 2

~~~python
def build_baseline_prompt_v2(instruction: str) -> str:
    return f"""\
You are a senior Python engineer.  Fulfill the following task by writing production-
    ready code.

**Task**:
{instruction.strip()}

**Requirements**:
- Python 3, include type hints
- One well-formed function or class with a descriptive name
- A docstring (inputs, outputs, edge cases)
- PEP8 style (4-space indent, snake_case)
- At least one unit test using `assert` or `unittest`

**Implementation**:
```python
"""

def build_rag_prompt_v2(instruction: str, retrieved: str) -> str:
    return f"""\
You are a senior Python engineer.  Use the retrieved examples to guide your
    implementation.

**Retrieved Examples**:
{retrieved.strip()}

**Task**:
{instruction.strip()}

~~~

```
**Requirements**:
- Python 3 with type hints
- Clean function or class design with a docstring
- Adhere to PEP8 conventions
- Include at least one unit test


**Implementation**:
‘‘‘python
"""
```

Listing 2: Python code for building prompts v2.

A.3. Prompt 6.2

```
def build_baseline_prompt_v6_2(instruction: str) -> str:
    return f"""# PYTHON LIBRARY-BASED CODE GENERATION TASK

YOUR OBJECTIVE IS TO GENERATE A **COMPLETE, SINGLE PYTHON 3 FILE** THAT DIRECTLY
SOLVES THE TASK DESCRIBED BELOW. THIS TASK REQUIRES SIGNIFICANT UTILIZATION OF A SPECIFIC
PYTHON LIBRARY, WHICH SHOULD BE INFERRED FROM THE TASK DESCRIPTION.

---

## TASK DESCRIPTION

{instruction.strip()}

---

## IMPLEMENTATION REQUIREMENTS & OUTPUT FORMAT

**CODE REQUIREMENTS:**
- THE OUTPUT MUST BE A SINGLE, RUNNABLE PYTHON 3 SCRIPT OR A SELF-CONTAINED MODULE.
- HEAVILY UTILIZE THE TARGET PYTHON LIBRARY’S APIS. YOUR SOLUTION SHOULD DEMONSTRATE
  EFFECTIVE USE OF THE LIBRARY.
- PRODUCE CLEAN, PRODUCTION-QUALITY PYTHON 3 CODE.
- INCLUDE TYPE HINTS FOR ALL FUNCTION SIGNATURES AND IMPORTANT VARIABLES.
- WRITE A CLEAR DOCSTRING FOR ANY PRIMARY FUNCTIONS OR CLASSES YOU DEFINE.
- ADHERE TO PEP 8 STYLE GUIDELINES.
- IF THE TASK INVOLVES CREATING A REUSABLE FUNCTION OR CLASS, DEMONSTRATE ITS
  USAGE WITHIN THE SCRIPT OR PROVIDE A SIMPLE UNIT TEST.
- ENSURE ALL NECESSARY IMPORTS FROM THE TARGET LIBRARY (AND STANDARD PYTHON LIBRARIES)
  ARE INCLUDED AT THE BEGINNING OF THE FILE.

**OUTPUT:**
- PROVIDE **ONLY** THE PYTHON CODE FOR THE SINGLE FILE.
- DO NOT INCLUDE ANY SURROUNDING TEXT, EXPLANATIONS, OR MARKDOWN FORMATTING
  (PYTHON COMMENTS AND DOCSTRINGS ARE ENCOURAGED WITHIN THE CODE).
- START YOUR CODE IMMEDIATELY FOLLOWING THE ‘‘‘PYTHON MARKER.

‘‘‘PYTHON
"""

def build_rag_prompt_v6_2(instruction: str, retrieved: str) -> str:
    retrieved_snippet = truncate_to_n_tokens(retrieved, 1800)
    return f"""# PYTHON LIBRARY-BASED CODE GENERATION TASK (WITH CONTEXT)

YOU ARE AN EXPERT PYTHON ENGINEER. YOUR OBJECTIVE IS TO GENERATE A **COMPLETE,
SINGLE PYTHON 3 FILE** THAT DIRECTLY SOLVES THE ’TASK DESCRIPTION’ BELOW.
THIS TASK REQUIRES SIGNIFICANT UTILIZATION OF A SPECIFIC PYTHON LIBRARY.
```

~~~
YOU ARE PROVIDED WITH 'RETRIEVED CODE EXAMPLES' FROM THIS LIBRARY TO GUIDE YOUR
    IMPLEMENTATION.

---

## RETRIEVED CODE EXAMPLES (CONTEXTUAL GUIDANCE)

```PYTHON
{retrieved_snippet.strip()}
"""
~~~

Listing 3: Python code for building prompts v6.2.

A.4. Prompt 6.3

~~~
def build_baseline_prompt_v6_3(instruction: str) -> str:
    """
    Uppercase variant of v6 baseline (con nota 'no errors' / clean code).
    """
    return f"""# PYTHON LIBRARY-BASED CODE GENERATION TASK

YOUR OBJECTIVE IS TO GENERATE A **COMPLETE, SINGLE PYTHON 3 FILE** THAT DIRECTLY
SOLVES THE TASK DESCRIBED BELOW. THIS TASK REQUIRES SIGNIFICANT UTILIZATION OF A
    SPECIFIC
PYTHON LIBRARY, WHICH SHOULD BE INFERRED FROM THE TASK DESCRIPTION.

---

## TASK DESCRIPTION

{instruction.strip()}

---

## IMPLEMENTATION REQUIREMENTS & OUTPUT FORMAT

**CODE REQUIREMENTS:**
- THE OUTPUT MUST BE A SINGLE, RUNNABLE PYTHON 3 SCRIPT OR A SELF-CONTAINED MODULE.
- HEAVILY UTILIZE THE TARGET PYTHON LIBRARY'S APIS. YOUR SOLUTION SHOULD DEMONSTRATE
  EFFECTIVE USE OF THE LIBRARY.
- PRODUCE CLEAN, PRODUCTION-QUALITY PYTHON 3 CODE.
- INCLUDE TYPE HINTS FOR ALL FUNCTION SIGNATURES AND IMPORTANT VARIABLES.
- WRITE A CLEAR DOCSTRING FOR ANY PRIMARY FUNCTIONS OR CLASSES YOU DEFINE.
- ADHERE TO PEP 8 STYLE GUIDELINES.
- IF THE TASK INVOLVES CREATING A REUSABLE FUNCTION OR CLASS, DEMONSTRATE ITS
  USAGE WITHIN THE SCRIPT OR PROVIDE A SIMPLE UNIT TEST.
- ENSURE ALL NECESSARY IMPORTS FROM THE TARGET LIBRARY (AND STANDARD PYTHON LIBRARIES
    )
  ARE INCLUDED AT THE BEGINNING OF THE FILE.
- THE SCRIPT MUST NOT GIVE ERRORS AND BE A HIGH-QUALITY CLEAN CODE

**OUTPUT:**
- PROVIDE **ONLY** THE PYTHON CODE FOR THE SINGLE FILE.
- DO NOT INCLUDE ANY SURROUNDING TEXT, EXPLANATIONS, OR MARKDOWN FORMATTING
  (PYTHON COMMENTS AND DOCSTRINGS ARE ENCOURAGED WITHIN THE CODE).
- START YOUR CODE IMMEDIATELY FOLLOWING THE ```PYTHON MARKER.

```PYTHON
"""

def build_rag_prompt_v6_3(instruction: str, retrieved: str) -> str:
    """
    Uppercase variant of v6 RAG (con nota 'no errors' / clean code).
    """
~~~

~~~
    retrieved_snippet = truncate_to_n_tokens(retrieved, 1800)
    return f"""# PYTHON LIBRARY-BASED CODE GENERATION TASK (WITH CONTEXT)

YOU ARE AN EXPERT PYTHON ENGINEER. YOUR OBJECTIVE IS TO GENERATE A **COMPLETE,
SINGLE PYTHON 3 FILE** THAT DIRECTLY SOLVES THE 'TASK DESCRIPTION' BELOW.
THIS TASK REQUIRES SIGNIFICANT UTILIZATION OF A SPECIFIC PYTHON LIBRARY.
YOU ARE PROVIDED WITH 'RETRIEVED CODE EXAMPLES' FROM THIS LIBRARY TO GUIDE YOUR
    IMPLEMENTATION. THE SCRIPT MUST NOT GIVE RRORS AND BE A HIGH-QUALITY CLEAN CODE

---

## RETRIEVED CODE EXAMPLES (CONTEXTUAL GUIDANCE)

```PYTHON
{retrieved_snippet.strip()}
"""
~~~

Listing 4: Python code for building prompts v6.3.

A.5. Prompt 7

```
def build_baseline_prompt_v7(instruction: str) -> str:
    return f"""\
# Python Code Generation Task (Library-Focused)

You are an expert programmer and Python engineer tasked with writing a **complete,
    standalone Python 3 script** that solves the problem described below.
This problem implicitly requires using a particular **Python library**, which must be
     inferred from the task.
Focus on demonstrating strong understanding and appropriate use of that librarys
    API.

---

## Task Description

{instruction.strip()}

---

## Code Requirements

- Produce a **self-contained** and **runnable** Python 3 script. It must be a high-
    quality and clean code.
- Make **extensive and idiomatic use** of the target library.
- Include all necessary `import` statements.
- Use **type hints** for all function signatures and relevant variables.
- Add **clear and concise docstrings** to main classes/functions.
- Follow **PEP 8** style and organize code into logical sections.
- If the task involves a reusable component (e.g., function/class), demonstrate its
    usage with a clear **main block** or **unit test**.
- Your solution should be **robust**, modular, and easy to understand.
- If you follow all the rules we've given you, you'll make us very proud and you
    might even become famous
---

## Output Format

- Output **only** the Python code.
- Do **not** include any explanation or markdown.
- Start writing your code exactly now

"""

```

~~~python
def build_rag_prompt_v7(instruction: str, retrieved: str) -> str:
    """
    v7 RAG     usa retrieved snippet come contesto.
    """
    retrieved_snippet = truncate_to_n_tokens(retrieved, 1800)

    return f"""\
# Python Code Generation Task (Library-Focused, with Context)

You are an expert Python developer. Your task is to write a **complete, single-file Python 3 script** that solves the problem described below.
This problem implicitly involves using a specific **Python library**, and to support your implementation, you are also given **retrieved code examples** from that library which it might help you.

---

## Task Description

{instruction.strip()}

---

## Retrieved Code Examples

Use the examples below as context. You may adapt patterns, API usage, and techniques shown here if relevant:

```python
{retrieved_snippet.strip()}

---

## Code Requirements

- The script must be **fully self-contained** and **runnable**. It must be a high-quality and clean code.
- Use the relevant librarys APIs **extensively** and **idiomatically**.
- Include all **necessary imports**.
- Add **type annotations** and well-written **docstrings**.
- Ensure clean code structure, following **PEP 8** style.
- If applicable, include a simple **main execution block** or **test case**.
- Emphasize clarity, reusability, and correctness.

---

## Output Format

- Provide **only** the Python code.
- No extra explanation, markdown, or commentary.
- Start writing your code exactly now

"""
~~~

Listing 5: Python code for building prompts v7.

A.6. Prompt 8

```python
def build_baseline_prompt_v8(instruction: str) -> str:
    return f"""\
# Library-Centric Python Code Generation (v8)

You are ChatGPT, a world-class Python engineer.

```

```
**Task**
Write a **single, self-contained Python 3 file** that completely solves the problem
    below.
The solution must make *substantial, idiomatic* use of the one Python library
    implicitly
required by the task.

---

## Problem

{instruction.strip()}

---

## Implementation Checklist
1. Plan first (internally)     do **not** output the plan.
2. Code quality
   - all necessary imports (standard + target library)
   - type hints everywhere
   - concise docstrings explaining each public items intent and library usage
   - PEP 8 compliance
3. Demonstration     ‘if __name__ == "__main__":‘ **or** minimal pytest-style test
4. Robustness     handle edge cases, raise informative errors
5. Clarity > cleverness

---

## Output Rules

- **Output ONLY the code** for the single file.
- No extra text or markdown.
- Begin code immediately after ‘ ‘‘‘python‘ and end with nothing else.

‘‘‘python
"""

def build_rag_prompt_v8(instruction: str, retrieved: str) -> str:
    retrieved_snippet = truncate_to_n_tokens(retrieved, 1800)

    return f"""\
# Library-Centric Python Code Generation with Context (v8)

You are ChatGPT, a world-class Python engineer.

You receive:
1. **Problem description** (requires a specific Python library).
2. **Retrieved code examples** from that library.

Craft a **single, self-contained Python 3 file** solving the problem, leveraging and
    adapting patterns from the examples.

---

## Problem

{instruction.strip()}

---

## Retrieved Library Examples

‘‘‘python
{retrieved_snippet.strip()}
"""
```

Listing 6: Python code for building prompts v8.

A.7. Prompt 9

```
def build_baseline_prompt_v9(instruction: str) -> str:
    return f"""#  Library-Centric Python Code Generation (v9     Baseline)

You are **a senior Python engineer**.
Your mission is to deliver a **single, self-contained Python 3 file** that *fully* solves the problem below, *making substantial and correct use of the one Python library implicitly required*.

---

## Problem

{instruction.strip()}

---

## Workflow (do not output steps 1  3  )

1. **Analyse** the task and *identify the target library* with high confidence.
2. **Plan** the solution in your head (modules, functions, data flow).
3. **Self-review checklist**
   - Every function/class is type-annotated and has a concise docstring.
   - Each external call **exactly matches an existing API** of the library
     (verify names, parameter order, return types).
   - No extraneous imports or unused variables.
   - PEP 8 compliance.
   - Provide a tiny runnable demo or ‘pytest‘-style test in a
     ‘if __name__ == "__main__":‘ guard.
   - Script runs without internet access or extra files.

4. **Write the code** (start now).

---

## Output rules

- **Output ONLY the Python code.**
- No explanations, markdown, or comments outside the source file.
- Begin immediately after the next line:

‘‘‘python
"""

def build_rag_prompt_v9(instruction: str, retrieved: str) -> str:
    retrieved_snippet = truncate_to_n_tokens(retrieved, 1800)
    return f"""# Library-Centric Python Code Generation with Context (v9     RAG)

You are **a senior Python engineer**.
Alongside the problem, you receive **verified code excerpts** from the relevant library.
Leverage them to craft a **single, runnable Python 3 file** that solves the task, re-using APIs and patterns where appropriate.

---

## Problem

{instruction.strip()}

```

```
---

## Retrieved Library Excerpts (grounding context)

‘‘‘python
{retrieved_snippet.strip()}
"""
```

Listing 7: Python code for building prompts v9.

References

[1] Shuo Ren, Daya Guo, Shuai Xu, Zhi Jin Zhou, Shujie Zhang, Shuo Ma, and Ming Zhou. Codebleu: a method for automatic evaluation of code synthesis. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*, pages 4505–4515, 2020.