CHORUS: Zero-shot Hierarchical Retrieval and Orchestration for Generating Linear Programming Code

Tasnim Ahmed[1[0000−0002−0799−1180]] and Salimur Choudhury[1[0000−0002−3187−112X]]

School of Computing, Queen's University, Ontario, Canada
{tasnim.ahmed, s.choudhury}@queensu.ca

Abstract. Linear Programming (LP) problems aim to find the optimal solution to an objective under constraints. These problems typically require domain knowledge, mathematical skills, and programming ability, presenting significant challenges for non-experts. This study explores the efficiency of Large Language Models (LLMs) in generating solver-specific LP code. We propose CHORUS, a retrieval-augmented generation (RAG) framework for synthesizing Gurobi-based LP code from natural language problem statements. CHORUS incorporates a hierarchical tree-like chunking strategy for theoretical contents and generates additional metadata based on code examples from documentation to facilitate self-contained, semantically coherent retrieval. Two-stage retrieval approach of CHORUS followed by cross-encoder reranking further ensures contextual relevance. Finally, expertly crafted prompt and structured parser with reasoning steps improve code generation performance significantly. Experiments on the NL4Opt-Code benchmark show that CHORUS improves the performance of open-source LLMs such as Llama3.1 (8B), Llama3.3 (70B), Phi4 (14B), Deepseek-r1 (32B), and Qwen2.5-coder (32B) by a significant margin compared to baseline and conventional RAG. It also allows these open-source LLMs to outperform or match the performance of much stronger baselines—GPT3.5 and GPT4 while requiring far fewer computational resources. Ablation studies further demonstrate the importance of expert prompting, hierarchical chunking, and structured reasoning.

Keywords: Linear Programming · Large Language Models · Code Generation · Retrieval-Augmented Generation · Gurobi Solver.

1 Introduction

Solving Linear Programming (LP) problems, an essential field of applied mathematics, has demonstrated value in areas like supply chain management, energy scheduling, marketing, resource allocation, networking, etc [1]. Traditionally, this process involves three steps: translating real-world scenarios into mathematical formulations by identifying variables, objectives, constraints, and parameters; encoding these formulations using modeling languages such as Python, R, or

C++; and finally, executing the optimization using solvers [2]. While practitioners proficient in mathematical modeling are generally capable of using solver APIs, the overall process still demands fluency in both the mathematical and programming domains.

Large Language Models (LLMs) have become powerful tools, demonstrating remarkable performance across various tasks, including code generation and reasoning tasks [3]. As LLMs grow in size, they serve effectively as standalone knowledge repositories, with facts encoded in their parameters [4], and can be further enhanced through fine-tuning on specific downstream tasks [5]. However, even large models often lack sufficient domain-specific knowledge for specialized tasks, and their factual accuracy can diminish as the world evolves. Moreover, updating a model's internal knowledge through fine-tuning or pretraining remains challenging, especially with vast and frequently changing text corpora, e.g., code or tool documentation [6]. To address these limitations, retrieval techniques such as Retrieval-Augmented Generation (RAG) have been introduced [7], augmenting model inputs by appending contextually relevant documents retrieved from external knowledge corpora. However, traditional RAG pipelines have limitations. LLMs face challenges processing numerous chunks (e.g., top-100), not only due to efficiency constraints but also because shorter top-k lists (e.g., 5 or 10) improve generation accuracy [8]. With a small k, ensuring high recall of relevant content is crucial. A retrieval model alone may struggle with local alignments across the embedding space, while a cross-encoding ranking model better selects top-k contexts from top-N candidates ($N \gg k$) [9]. Furthermore, fixed-size chunking in code documentation disrupts dependencies, breaking functions, losing syntax, and separating explanations, reducing comprehensibility and leading to ineffective retrieval with incomplete functions and missing variables, increasing hallucination risks.

Efforts to simplify mathematical programming (linear, nonlinear, etc.) using LLMs aim to enhance accessibility for non-experts. Research studies have largely focused on utilizing LLMs to generate mathematical formulations and end-to-end solutions, converting problem descriptions into solver code. Notable contributions in generating mathematical formulation from LP problem descriptions include NL4Opt [2] and LM4OPT [10]. The NL4Opt competition explored converting natural language into structured LP formulations. The dataset samples from this competition include an input paragraph describing an LP problem, with annotations for entity extraction and semantic parsing to generate the representation of the objective and constraints. Using the NL4Opt dataset, the LM4OPT framework was proposed to enhance the performance of smaller LLMs in generating mathematical formulations. The methodology focuses on progressive fine-tuning, starting with broader domain contexts before specializing in formulation task-specific datasets. Tsouros et al. [11] proposed one of the first end-to-end solutions for this task, introducing an LLM-based system that generates mathematical formulations and solver codes from prompts. While demonstrating promise on the NL4Opt dataset, it lacked benchmark comparisons and relied solely on pretrained LLMs. E-OPT [12] provided a benchmark evaluating

the mathematical programming code generation capabilities of LLMs across linear, nonlinear, and tabular problems. Teshinizi et al. [13] introduced OptiMUS, a a multi-agent LLM framework to both formulate and solve mixed-integer LP problems based on natural language descriptions. It iteratively engages specialized agents, generating multiple responses per problem, leading to very high computational overhead. Another work, OptLLM [14] integrates LLMs with external solvers to automate the modelling and solving of LP problems. The authors curated a dataset with 100 problems from the NL4Opt development set to evaluate the performance of GPT3.5 [15], GPT4 [16], and their fine-tuned Qwen model. In addition, OptiGuide [17] combines GPT4 with plain-text user queries to generate insights into optimization outcomes which enhances decision-making in supply chain management. Researchers have explored LLM-based approaches that solve such problems without external solvers. Yang et al. [18] introduced OPRO, a framework that iteratively refines solutions using a meta-prompt. However, LLMs are not well-suited for such tasks, as their generation relies primarily on next-token prediction rather than structured mathematical reasoning. To summarize, recent advancements have demonstrated that numerous researchers are actively working on generating problem formulations or solver code from LP or similar mathematical programming problem descriptions using LLMs. However, solver codes typically rely heavily on documentation, and popular solvers frequently update their frameworks or tool versions to incorporate new features. These solvers often function as APIs that support multiple programming languages. As a result, fine-tuning a model on a specific dataset for a particular solver ties the model to that specific solver version and programming language. A major version update in the future could render the generated codes prone to errors, necessitating further annotation and re-fine-tuning, a resource-intensive process. Thus, depending on the parametric knowledge of LLMs or fine-tuning for code generation can lead to rigid, narrowly focused solutions that require significant effort to maintain over time. Moreover, while LLMs excel in traditional analytical and algorithmic code generation due to abundant examples and libraries, the scarcity of solver-specific LP code examples limits their domain-specific knowledge, a challenge for both pretrained and fine-tuned LLMs [13].

To address this limitation, our study focuses on providing an adaptive end-to-end solution to generate solver code directly from mathematical problem descriptions, without relying on any specific solver or LLM. We hypothesize that providing an enhanced context is essential to tackle this complex task. Therefore, in this paper, we propose a novel RAG-based framework, CHORUS (Contextual Hierarchical Orchestration for Retrieval-aUgmented code Synthesis), to address this challenge. To the best of our knowledge, this is the first work to introduce a retrieval mechanism for generating optimization solver code directly from natural language problem descriptions. In this study, we generate LP codes for the Gurobi [19] solver API in Python, widely used for its efficiency and robust optimization algorithms. However, our approach remains adaptable to any mathematical problem, given an available solver. Our contributions are as follows:

1. We introduce CHORUS, a novel zero-shot RAG framework that enhances LLMs' retrieval abilities by leveraging hierarchical chunking and contextual metadata from LP solver (Gurobi) documentation for code generation.
2. Proposed two-stage retrieval process with cross-encoder reranking improves code generation accuracy significantly.
3. Expertly crafted prompt and structured parser with reasoning steps improve the reasoning capability of LLMs as well as facilitates automated inference, execution and evaluation.
4. Experiments on the NL4Opt-Code benchmark show significant gains over baseline LLMs and traditional RAG, allowing open-source LLMs to achieve GPT4-level performance with lower computational costs.

2 CHORUS

This section delineates CHORUS, which enhances LLM retrieval by leveraging hierarchical chunking and contextual metadata from LP solver documentation. It describes the retrieval process with cross-encoder reranking and details the use of an expertly crafted prompt with a structured parser and reasoning steps. Figure 1 presents an overview of the proposed pipeline at inference.

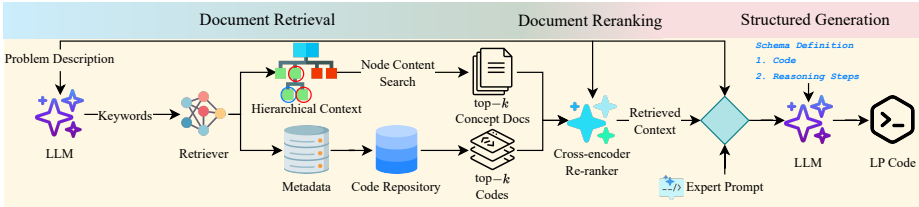

Fig. 1: **CHORUS framework for generating Gurobi solver code from LP descriptions.** It retrieves and reranks relevant conceptual and implementation-level documents, supplements them with an expert prompt, and employs a structured output parser for automated execution and improved reasoning.

2.1 Document Structuring and Indexing

The Gurobi documentation, serving as the primary source of external context, exhibits a dual nature. The first category comprises theoretical and contextual content that includes descriptions of various models, variables, and constraint types, as well as code snippets. This content is inherently rich in semantic information and provides a deep theoretical foundation necessary for understanding the nuances of different mathematical programming problem formulations. The second category consists solely of complete code examples that demonstrate the practical application of the Gurobi library in solving diverse mathematical programming scenarios. Although these code examples lack explanatory text, they embody critical implementation details that are essential for generating syntactically correct and functionally appropriate code. Given the heterogeneity in the types of content, we adopt a dual approach for storing and retrieving documents.

Hierarchical Tree Indexing of Theoretical Content Technical documents are inherently structured, and flattening it into equal-length chunks disrupts semantic coherence. By modeling the documentation as a tree, CHORUS mimics human navigation patterns, starting from broad overviews and drilling down to specifics. In a recent study on multi-document QA, Chen et al. [20] proposed a hierarchical multi-route retrieval system that enhances retrieval accuracy and helps reduce hallucinations. Our manual analysis of the retrieved content supports this idea, suggesting that such a retrieval approach helps minimize hallucination by providing the LLM with self-contained and contextually complete technical information. The theoretical documents are processed using the inherent structure of the source PDFs. We extract metadata and content from the PDF files to construct a hierarchical tree representation. At the root of this tree is the entire document, which is then segmented into chapters. Each chapter is further divided into sections, and these sections are subdivided into subsections. The hierarchical tree is designed to preserve both detailed and high-level contextual information. The main bulk of the theoretical content is distributed across the leaf nodes, while the intermediary nodes preserve introductory texts that usually contain summaries or high-level definitions of their child nodes. Each node is considered a chunk.

Metadata-Augmented Indexing of Code Examples Directly searching code syntax for natural language queries suffers from a vocabulary mismatch. Gao et al. [21] introduced HyDE (Hypothetical Document Embeddings), a dense retrieval method that generates an ideal document, embeds it into a vector, and retrieves similar real documents. This approach enhances retrieval precision by capturing the conceptual essence and mitigating vocabulary mismatches. Inspired by this workflow, our proposed code example retrieval mechanism depends upon enriched metadata to bridge the semantic gap between natural language queries and stored code examples. In contrast to the theoretical contents, the complete code examples are managed as individual chunks, each representing a distinct code snippet. These snippets are stored in a vector database along with generated metadata. For every code example, we generate a set of domain-agnostic keywords $(5 - 7)$ using an LLM providing an expert prompt. These keywords capture salient features such as the type of variables, the nature of the problem, and the specific constraints used in the model which abstract the functionality of the code. Additionally, a concise 2-3 line natural language synopsis is generated for each code snippet using the same approach. This metadata allows semantic alignment with user queries without requiring direct lexical matches to code syntax, which is critical for interdisciplinary problems where users describe constraints in non-technical terms.

The impact of our document structuring methodology is demonstrated through comparative analyses of chunk characteristics. Figure 2 highlights the natural variation in chunk lengths between theoretical documents and code examples, contrasting with the rigid fixed-length chunking used in traditional RAG systems. Figure 3 validates our metadata augmentation strategy through lexical

analysis. The raw code term frequencies reveal dominance of implementation-specific tokens (e.g., "gurobipy", "batchid", "continue") that rarely appear in natural language queries. Conversely, the metadata-derived terminology demonstrates improved alignment with natural query patterns.

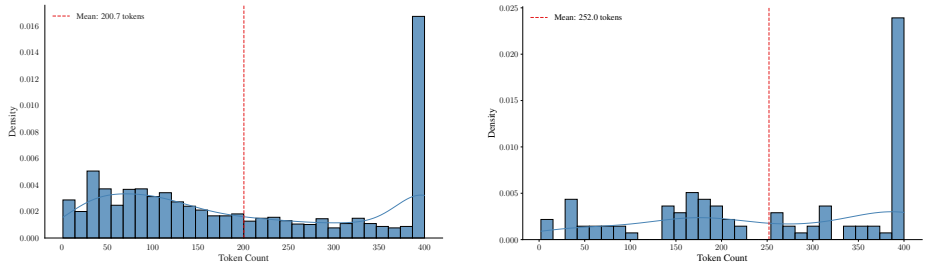

Fig. 2: Token length distributions for (left) theoretical documentation chunks and (right) code examples.

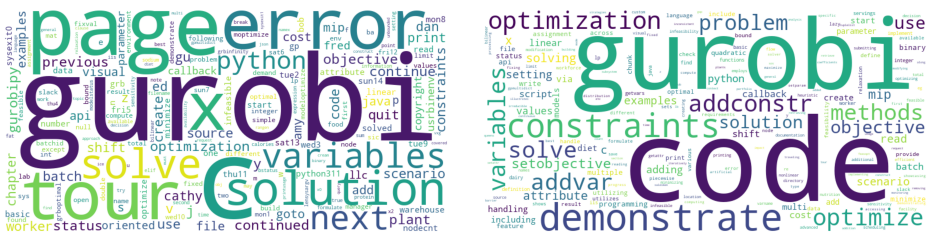

Fig. 3: Word frequency analysis contrasting (left) raw code token prevalence with (right) curated metadata terminology from summaries and keywords.

2.2 Two-Stage Context Retrieval

The retrieval process operates in two phases: conceptual context retrieval from the hierarchical documentation and code example retrieval from the indexed implementations. This separation allows the framework to retrieve both theoretical foundations and practical examples.

Hierarchical Context Retrieval An LLM starts with extracting a set of keywords from the problem description, focusing on LP-specific concepts (e.g., "binary variables," "supply chain optimization"). Keyword extraction is guided by a prompt engineered to prioritize terms likely to appear in technical documentation. These keywords are used to traverse the hierarchical document tree. For each keyword, top-k most relevant nodes are selected based on the cosine

similarity between their embeddings and node content. The retrieved chunk post-processing follows a dynamic, context-aware strategy. To ensure retrieved chunks are contextually self-contained yet concise, adaptive chunk construction dynamically combines sibling or parent content when node sizes fall below the maximum chunk size threshold (400 tokens). For smaller nodes, sibling content is concatenated iteratively until the token limit is reached. Crucially, parent node summaries are prepended to child content to retain hierarchical context. For example, a subsection describing quadratic constraints inherits the introductory context from its parent subsection on constraints, which ensures retrieved chunks provide both specificity and conceptual grounding. This approach mirrors human navigation patterns in technical documents, where practitioners iteratively narrow from broad overviews to granular details.

Code Example Retrieval The second phase of the retrieval process focuses on the complete code examples. Here, the keywords extracted from the problem description are matched not directly against the code itself but against the metadata that accompanies each complete code example. A similarity search is conducted within the vector database to retrieve top m candidate examples that best match descriptive keywords. The reliance on metadata rather than raw code ensures that the retrieval process remains robust to variations in coding style and syntax. This approach avoids lexical mismatches between natural language queries (e.g., "minimize delivery costs") and code variables (e.g., `model.setObjective(sum(c[i,j]*x[i,j] for i,j in routes), GRB.MINIMIZE)`).

2.3 Cross-Encoder Reranking for Contextual Relevance

After the initial retrieval phase, the system retains two candidate sets: one comprising $k \times n_{keywords}$ conceptual documents from the hierarchical tree and the other comprising m complete code examples. However, to further refine the selection and ensure that only the most relevant documents are utilized during code generation, we introduce a re-ranking mechanism based on a cross-encoder model [22]. While dual-encoders excel at scalable retrieval, their separate encoding of queries and documents limits their ability to capture fine-grained semantic relationships. The cross-encoder model is trained on the MS MARCO passage ranking task[1] and such models have proven to transfer very effectively to new domains, including technical text and code. The re-ranking process involves constructing input pairs for the cross-encoder, where each pair consists of the user query, keywords, and the text content (with metadata, in the case of code examples) of a candidate document. The cross-encoder model then computes a relevance score for each pair, effectively quantifying the semantic alignment between the query and the document. The candidate documents are subsequently sorted in descending order of their scores, and a selective truncation is performed wherein the top-3 conceptual documents and the top-2 code examples are retained for further processing.

[1] github.com/microsoft/MSMARCO-Passage-Ranking

2.4 LLM Code Generation

The final set of retrieved and re-ranked documents, comprising both conceptual context and code examples, is then integrated into the expert prompt that is fed to the language model. This integration serves a dual purpose—it enriches the input with domain-specific knowledge and ensures that the generated code adheres to both theoretical principles and practical implementation guidelines as prescribed by the Gurobi documentation.

Expert Prompting To ensure that the generated code rigorously matches each problem specification while maintaining high-quality coding practices, we implement specialized system and user prompts. The system prompt directs the language model to assume the role of an expert in both operations research and Python-based Gurobi development, whereas the user prompt outlines strict requirements such as the function name, error handling strategy, variable naming conventions, and constraint modeling rules. These prompts merge specialized domain guidance with explicit style and accuracy rules, which enforce the LLM to create Python functions that accurately capture the mathematical formulation, limit unneeded output, and include exception handling for potential errors.

Structured Output with Improved Reasoning A fundamental requirement of CHORUS is its capacity for end-to-end automation, including code inference, evaluation, and execution without reliance on external annotations or human intervention. Preliminary experiments with LLMs revealed a notable inconsistency: despite configuring the sampling temperature to minimize stochasticity, model responses demonstrate significant variability across queries. This observation shows a contradictory strain between the unstructured nature of responses—rooted in their training on vast, heterogeneous corpora—and the structured precision demanded by code generation tasks. To enforce consistency, we implemented a structured output-parsing mechanism, wherein each LLM response is validated against a predefined schema (e.g., a class definition) to extract executable code. However, as demonstrated in [23], strict formatting constraints can impair LLMs' performance on analytical and reasoning tasks, with degradation intensifying as constraints tighten. We assume that enforcing code-only outputs, devoid of any explanatory text, may inadvertently restrict the model's capacity to engage in stepwise reasoning—a critical factor in ensuring code correctness. This hypothesis also aligns with empirical findings from OpenAI[2], which suggest that LLM output quality improves when models are instructed to provide intermediate reasoning steps within a structured schema. Building on this insight, we augmented our output schema with a dedicated `reasoning_steps` field, explicitly instructing the model to validate the code alignment with the problem specification (e.g., variable definitions, constraint formulations, and objective functions). The schema definition is as follows:

[2] openai.com/index/introducing-structured-outputs-in-the-api

```
class GurobiSolution(BaseModel):
    code: str = Field(description="Complete Python code using
    Gurobi API that solves the LP problem")
    reasoning_steps: str = Field(description="Justification
    mapping code components to problem requirements")
```

3 Experiments

3.1 Task Formulation

Given a natural language problem description X, the CHORUS framework implements a two-stage retrieval process to identify relevant context from structured documentation and code examples. The hierarchical retriever R_H extracts conceptual documents $D_C = \{d_{c1}, ..., d_{ck}\}$ by traversing a tree-structured index of theoretical content, preserving parent-child context relationships. Simultaneously, the metadata-augmented retriever R_M identifies code examples $D_E = \{d_{e1}, ..., d_{em}\}$ through keyword and summary alignment with X. These candidates are reranked by a cross-encoder C to yield optimal context subsets $D_C^* \subseteq D_C$ and $D_E^* \subseteq D_E$. The generator G, conditioned on X, D_C^*, and D_E^*, produces structured output $Y_{\text{structured}}$ containing:

$$Y_{\text{structured}} = \langle \text{code}, \text{reasoning_steps} \rangle$$

where:

- code: Executable Gurobi code adhering to API constraints
- reasoning_steps: Natural language justification mapping code to X

3.2 Dataset and Metrics

To the best of our knowledge, there are currently no publicly available datasets on general LP problems except the data from the NL4Opt competition. We conducted an empirical study using the NL4Opt-Code dataset, which we curated from the original problem descriptions found in the NL4Opt dataset. The NL4Opt dataset includes LP Word Problems (LPWPs) across six domains: sales, advertising, investment, production, transportation, and sciences. Each LPWP has problem descriptions and annotations for key entities such as constraints, objectives, and variables. We expanded this dataset by generating Gurobi code for each LPWP. The NL4Opt-Code dataset now contains LPWPs, each paired with its corresponding Gurobi solver code.

Dataset Annotation The dataset annotation was performed by a domain expert (a graduate student who completed an undergraduate course in Optimization and a graduate course in Combinatorial Optimization). The annotation process was carried out in two stages. In the first stage, the annotator derived the LP formulation for each problem, ensuring that the mathematical representation accurately captured the nuances of the problem description. These formulations were then verified by another expert with similar qualifications to ensure their correctness. In the second stage, the domain expert wrote the Gurobi code based on the verified formulations, which was then double-checked by another expert.

Evaluation Metrics The evaluation metric used in E-OPT [12] uses the output code to produce numerical answers for variables and objectives, which are compared against the ground truth for accuracy. On the other hand, NL4Opt compares the coefficients of variables and decision variables in constraints and the objective function against the ground truth based on an intermediate matrix representation. However, this approach imposes certain limitations, such as requiring all constraints to be in the less-than-or-equal-to form and the objective function to be in the maximization form. As a result, a correct formulation may be considered incorrect if it does not conform to these specific rules. Unlike NL4Opt and ReSocratic, we adopted a more stringent definition for our primary evaluation metric, "accuracy." In addition, we incorporated three supplementary measures that evaluate code quality at varying levels of detail—specifically, "syntactic validity," "semantic similarity," and "edit distance."

Accuracy: A generated response is considered correct only if the optimal objective function value from the generated solver code matches that of the reference solver code. This metric offers an automated and fast method for evaluating the solution, eliminating the need for human evaluators. Additionally, when the objective value matches, it generally indicates that all constraints and objective functions are correctly aligned.

$$\text{Accuracy} = \frac{1}{|X|} \sum_{x \in X} \mathbb{I}\big(f\big(Y_{\text{gen}}(x)\big) = f\big(Y_{\text{ref}}(x)\big)\big) \tag{1}$$

Here,
- X be the set of all problem instances in the test set.
- $Y_{\text{gen}}(x)$ be the generated code for problem instance $x \in X$.
- $Y_{\text{ref}}(x)$ be the reference code for problem instance $x \in X$.
- f be the optimal objective function value produced by executing a solver code.
- $\mathbb{I}$ be the indicator function, where $\mathbb{I}(A) = 1$ if condition A is true, and $\mathbb{I}(A) = 0$ otherwise.

Syntactic Validity: It checks whether the generated code can be parsed and executed without syntax errors. It serves as a sanity check to ensure that the model produces well-formed Python code.

$$\text{Syntactic Validity} = \frac{1}{|X|} \sum_{x \in X} \mathbb{I}\big(\text{parse}\big(Y_{\text{gen}}(x)\big) = \text{True}\big) \tag{2}$$

Semantic Similarity: It quantifies the conceptual similarity between the generated and reference code using embeddings from a pre-trained language model (`all-MiniLM-L6-v2`[3]). It is calculated using cosine similarity between the embedded representations. Jin et al. [24] demonstrated that traditional metrics such as BLEU and BERTScore frequently fall short in capturing nuanced semantic meaning and struggle with domain-specific terminology common in code, whereas LLM-based embeddings show a much stronger alignment with human evaluations.

[3] huggingface.co/sentence-transformers/all-MiniLM-L6-v2

$$\text{Semantic Similarity} = \frac{1}{|X|} \sum_{x \in X} \cos\big(\text{embed}\big(Y_{\text{gen}}(x)\big),\ \text{embed}\big(Y_{\text{ref}}(x)\big)\big) \tag{3}$$

Edit Distance: This similarity metric measures the character-level similarity between the generated and reference code using the Levenshtein ratio. Tree-based edit distances closely correlate with execution accuracy and often offer fine-grained insights into code similarity across languages [25].

$$\text{Edit Distance} = \frac{1}{|X|} \sum_{x \in X} \text{SequenceMatcher}(Y_{\text{gen}}(x),\ Y_{\text{ref}}(x)) \tag{4}$$

3.3 Experimental Setup

We designed the evaluation framework for efficient execution on a local server. To achieve this, we intentionally excluded methodologies that depend on online context and instead relied on open-source models and retrieval processes. Some entities may prefer not to use closed-source online LLMs for security and privacy. In certain scenarios, a solution may need to run on local devices or in remote areas. For instance, a recent country-wide internet outage in a country lasted several days, and if a business or operation depends on LP solving, a standalone local solution can be crucial.

Choice of LLMs We chose a diverse set of LLMs—Llama3.1 (8B), Llama3.3 (70B) [3], Phi4 (14B) [26], Deepseek-r1 (32B) [27], and Qwen2.5-coder (32B) [28]—and compared them with the well-known baselines, GPT3.5 and GPT4. This selection covers a wide range of parameter sizes and architectures, letting us explore the trade-offs between model complexity, computational efficiency, and code generation quality. For instance, Llama3.1 (8B) gives us insight into how lightweight models perform in resource-limited settings, while Llama3.3 (70B) serves as a benchmark for high-capacity models with stronger reasoning capabilities. Phi4 (14B) and Deepseek-r1 (32B) were selected for their unique training approaches that may enhance reasoning, and Qwen2.5-coder (32B), with its coding focus, is particularly suited for our code generation task. We acknowledge the importance of testing CHORUS with GPT-4 and GPT-3.5. However, our experimental focus on open-source models aligns with the core goal of this work: improving resource-efficient, locally deployable LLMs that support accessibility, privacy, and computational feasibility for a wide range of users. Although GPT-4 and GPT-3.5 perform well, their proprietary nature, high operational costs, and dependence on cloud-based APIs make them less practical for many real-world applications. Additionally, GPT series models' larger parameter size and inference demands make it significantly computationally expensive compared to the open-source models tested [29].

Hardware Configuration The experiments were conducted on a Linux server equipped with an Intel Xeon Platinum 8358 processor, featuring 38 MB of cache,

32 cores, and a maximum clock speed of 3.2 GHz. An NVIDIA H100 NVL GPU further accelerated the computational tasks with 95.8 GB of GPU memory. The system was supported by 251 GB of RAM.

3.4 Results

Table 1 presents a comparative analysis of the performance of the baseline models and the CHORUS framework across multiple LLMs. While GPT3.5 and GPT4 serve as baseline references, their strong performance (achieving accuracy scores of 0.5260 and 0.6367, respectively) emphasizes the effectiveness of closed-source models in LP code generation. Furthermore, the performance of GPT3.5 and GPT4 suggests that open-source LLMs (e.g., Qwen2.5-coder (32B)) are approaching the baseline performance of commercial, large-scale models. Among the open-source LLMs, the CHORUS framework results in substantial improvements in accuracy relative to the baseline. For example, the accuracy of Llama3.3 (70B) increases from 0.2289 to 0.5675, while Phi4 (14B) improves from 0.1938 to 0.6125. Notably, even smaller or mid-range models (e.g., Llama3.1 (8B) and Deepseek-r1 (32B)) demonstrate significant performance gains, which indicates that the incorporation of retrieval and re-ranking steps effectively enhances the alignment of generated code with the correct objective function and constraints. With the exception of the smallest model in this set (Llama3.1 (8B)), all open-source LLMs surpass GPT3.5 following the integration of the CHORUS framework, achieving performance levels comparable to GPT4. Given that GPT4 is considerably more computationally expensive and energy-intensive, the results demonstrate the effectiveness of CHORUS in optimizing the performance of locally deployed, resource-efficient LLMs. Syntactic validity improves in all CHORUS-based runs, indicating that a more coherent technical context reduces parsing errors. Results show that for the baseline open-source LLMs, the syntactic validity score ranges from 0.69 to 0.96. However, incorporating the CHORUS framework improves the score to 1.00, except for Llama models, where the score is in the range of 0.98. This demonstrates that integrating theoretical documents and code examples enhances syntactic correctness by ensuring the generated code is almost always valid. Such an improvement is particularly significant for code generation tasks, where syntactic accuracy directly impacts downstream usability and execution reliability. Likewise, semantic similarity improves with CHORUS, which implies that the integrated code examples and conceptual documentation help preserve semantic alignment between the generated and reference solutions. However, edit distance, which measures token-level similarity, is not a particularly reliable metric for evaluating code generation tasks. Our experimental results show that edit distance remains nearly similar between the baseline and CHORUS-based runs, despite significant improvements in accuracy. This is because a given problem can be implemented in multiple valid ways, leading to high edit distance scores even for correct solutions. Overall, our evaluation indicates that accuracy is the most critical metric for this task, as it closely mirrors the real-world performance of such a framework in practical applications.

Table 1: Experimental results for both Baseline and CHORUS frameworks across various LLMs. The parameter sizes of the LLMs are detailed in subsection 3.3.

Approach	Model	Accuracy (↑)	Syntactic Validity (↑)	Semantic Similarity (↑)	Edit Distance (↑)
Baseline	Llama3.1	0.0796	0.8581	0.8365	0.2008
	Llama3.3	0.2289	0.6920	0.9092	0.4185
	Phi4	0.1938	0.2872	0.9153	0.3921
	Deepseek-r1	0.1073	0.9654	0.9186	0.3276
	Qwen2.5-coder	0.4644	0.8618	0.9280	0.4728
	GPT3.5	0.5260	1.0000	0.9159	0.3933
	GPT4	0.6367	0.9931	0.8721	0.3565
CHORUS	Llama3.1	0.1349	0.9827	0.7941	0.2019
	Llama3.3	0.5675	0.9827	0.9291	0.4255
	Phi4	0.6125	1.0000	0.9379	0.4953
	Deepseek-r1	0.5848	1.0000	0.9141	0.3946
	Qwen2.5-coder	0.5986	1.0000	0.9367	0.4747

3.5 Ablation Study

We conduct a comprehensive ablation analysis to quantify the contributions of the core components of our proposed framework—expert prompting, retrieval, and structured reasoning. Table 2 compares five system configurations across five LLMs, which reveals critical insights into optimization code generation dynamics. All configurations use structured output schema for automated evaluation.

Impact of Expert Prompting The efficiency of expert prompting exhibits significant model-size dependence. While the Deepseek-r1 (32B) model achieves a 386.95% accuracy improvement when using expert prompts (rising from 10.73% to 52.25%), the smaller Llama3.1 (8B) model suffers a 61.3% reduction in accuracy (7.96% to 3.08%). Manual analysis of generated code samples reveals that this divergence derives from differences in instruction-following capacity and parametric knowledge. Larger models demonstrate superior adherence to the implementation and formatting rules of the expert prompt. Expert prompt occupies a significant portion of the context window in smaller models, which overwhelms their limited capacity, leading to syntax errors.

Traditional RAG Limitations Traditional fixed-length chunking strategy reduces accuracy by 46.14–89.33% compared to CHORUS. This reduction primarily arises from two systemic flaws inherent to traditional retrieval approaches. First, most of retrieved chunks contain fragmented or incomplete contents. Second, semantic dispersion introduces irrelevant content—the majority of the retrieved chunks mismatch problem domains, confusing models with conflicting API examples. These issues disproportionately affect smaller architectures i.e., accuracy of Llamma3.1 plummets to 1.4% with traditional RAG versus 13.5%

Table 2: **Ablation Results (Accuracy):** Structured reasoning provides consistent gains across LLMs. Traditional RAG underperforms CHORUS by 46.14–89.33%.

Configuration	Llama3.1	Llama3.3	Phi4	Deepseek-r1	Qwen2.5-coder
Baseline	0.0796	0.2289	0.1938	0.1073	0.4644
Baseline + Expert Prompt	0.0308	0.4833	0.5536	0.5225	0.5640
Traditional RAG	0.0144	0.1578	0.1698	0.1168	0.3224
CHORUS (w/o reasoning)	0.0104	0.5497	0.6020	0.5613	0.5721
CHORUS	0.1349	0.5675	0.6125	0.5848	0.5986

in CHORUS. Even coding-specialized models like Qwen2.5-coder (32B) exhibit 46.14% reduced accuracy, which demonstrates that domain expertise cannot compensate for poor context selection.

Impact of Structured Reasoning Omitting the `reasoning_steps` field from CHORUS's structured definition schema reduces accuracy by 1.71–92.29% across model scales. The reasoning field forces models to explicitly map problem elements to code components, a process that improves constraint coverage. This aligns with chain-of-thought [30] but extends its benefits to structured code synthesis through schema enforcement.

4 Conclusion

In this work, we introduced **CHORUS**, a RAG framework for generating Gurobi-based LP code from natural language descriptions. Our methodology ensures semantic cohesion in theoretical documentation through hierarchical chunking and bridges vocabulary mismatches with metadata-augmented retrieval of code examples. Experimental results on the NL4Opt-Code dataset demonstrate that CHORUS consistently outperforms baseline LLMs. Ablation studies further highlight the importance of core components of our proposed framework. One of the key strengths of CHORUS is its flexibility, as it remains both LLM-agnostic and solver-independent. While our experiments focus on generating LP code for Gurobi, the framework can support various mathematical problems, as long as a corresponding solver is available. By decoupling retrieval-augmentation from solver-specific logic, the use of CHORUS allows for seamless adaptation to future solver updates and emerging LLMs, ensuring continuous improvements over standard code generation methods.

4.1 Limitations

Although our framework demonstrates strong performance improvement compared to conventional RAG, certain limitations persist. Code generation remains highly sensitive to prompt engineering, and smaller models struggle to fully incorporate all contextual elements within limited context windows. While our methodology is solver-independent, its performance depends on the quality and relevance of the indexed documentation, a factor that varies across solvers. Ad-

ditionally, the alignment of LLMs with other optimization topics (e.g., integer linear, mixed, or non-linear problems) is left for future research. Another important consideration is the evolving nature of solver documentation. As solvers frequently introduce new methods and modify existing APIs, this can impact the consistency and completeness of the CHORUS knowledge base. To maintain system accuracy, the documentation parsing and knowledge base construction process would need to be re-executed periodically to reflect these updates. Despite these constraints, CHORUS lays a strong foundation by offering a resource-efficient alternative that allows open-source LLMs to approach GPT4-level performance.

4.2 Future Work

We believe that further exploration into adaptive retrieval strategies, multi-step reasoning, and agentic approach will open avenues for increasingly robust and domain-adapted code synthesis in optimization workflows. In future work, the CHORUS pipeline can be extended by incorporating advanced prompt engineering techniques such as role prompting, reflexion, ReAct, chain-of-thought, and chain-of-experts prompting [31]. Additionally, integrating few-shot learning strategies may further improve model performance in low-resource or domain-specific settings. Furthermore, the generalizability of the framework can be evaluated by applying it to additional mathematical optimization domains and across a broader set of solvers, including OR-Tools and IBM CPLEX.

References

1. Zhang, J., Wang, W., Guo, S., Wang, L., Lin, F., Yang, C., Yin, W.: Solving General Natural-Language-Description Optimization Problems with Large Language Models, (2024). arXiv: 2407.07924 `[math.OC]`.
2. Ramamonjison, R., Yu, T., Li, R., Li, H., Carenini, G., Ghaddar, B., He, S., Mostajabdaveh, M., Banitalebi-Dehkordi, A., Zhou, Z., Zhang, Y.: NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions. In: Ciccone, M., Stolovitzky, G., Albrecht, J. (eds.) Proceedings of the NeurIPS 2022 Competitions Track. Proceedings of Machine Learning Research, pp. 189–203. PMLR (2022)
3. Dubey, A. *et al.*: The Llama 3 Herd of Models, (2024). arXiv: 2407.21783 `[cs.AI]`.
4. Kandpal, N., Deng, H., Roberts, A., Wallace, E., Raffel, C.: Large Language Models Struggle to Learn Long-Tail Knowledge. In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) Proceedings of the 40th International Conference on Machine Learning. Proceedings of Machine Learning Research, pp. 15696–15707. PMLR (2023)
5. Roberts, A., Raffel, C., Shazeer, N.: How Much Knowledge Can You Pack Into the Parameters of a Language Model? In: Webber, B., Cohn, T., He, Y., Liu, Y. (eds.) Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 5418–5426. Association for Computational Linguistics, Online (2020)
6. Mitchell, E., Lin, C., Bosselut, A., Manning, C.D., Finn, C.: Memory-Based Model Editing at Scale. In: Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., Sabato, S. (eds.) Proceedings of the 39th International Conference on Machine

Learning. Proceedings of Machine Learning Research, pp. 15817–15831. PMLR (2022)
7. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems, pp. 9459–9474. Curran Associates, Inc. (2020)
8. Xu, P., Ping, W., Wu, X., McAfee, L., Zhu, C., Liu, Z., Subramanian, S., Bakhturina, E., Shoeybi, M., Catanzaro, B.: Retrieval meets Long Context Large Language Models. In: The Twelfth International Conference on Learning Representations (2024)
9. Yu, Y., Ping, W., Liu, Z., Wang, B., You, J., Zhang, C., Shoeybi, M., Catanzaro, B.: RankRAG: Unifying Context Ranking with Retrieval-Augmented Generation in LLMs. In: The Thirty-eighth Annual Conference on Neural Information Processing Systems (2024)
10. Ahmed, T., Choudhury, S.: LM4OPT: Unveiling the Potential of Large Language Models in Formulating Mathematical Optimization Problems, (2024). arXiv: 2403.01342 [cs.CL].
11. Tsouros, D.C., Verhaeghe, H., Kadiouglu, S., Guns, T.: Holy Grail 2.0: From Natural Language to Constraint Models. ArXiv **abs/2308.01589** (2023)
12. Yang, Z., Huang, Y., Shi, W., Feng, L., Song, L., Wang, Y., Liang, X., Tang, J.: Benchmarking LLMs for Optimization Modeling and Enhancing Reasoning via Reverse Socratic Synthesis, (2024). arXiv: 2407.09887 [cs.LG].
13. AhmadiTeshnizi, A., Gao, W., Udell, M.: OptiMUS: Optimization Modeling Using mip Solvers and large language models. ArXiv **abs/2310.06116** (2023)
14. Zhang, J., Wang, W., Guo, S., Wang, L., Lin, F., Yang, C., Yin, W.: Solving General Natural-Language-Description Optimization Problems with Large Language Models. In: Yang, Y., Davani, A., Sil, A., Kumar, A. (eds.) Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track), pp. 483–490. Association for Computational Linguistics, Mexico City, Mexico (2024)
15. Ye, J., Chen, X., Xu, N., Zu, C., Shao, Z., Liu, S., Cui, Y., Zhou, Z., Gong, C., Shen, Y., Zhou, J., Chen, S., Gui, T., Zhang, Q., Huang, X.: A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models, (2023). arXiv: 2303.10420 [cs.CL].
16. OpenAI, Achiam, J., Adler, S., *et al.*: GPT-4 Technical Report, (2024). arXiv: 2303.08774 [cs.CL].
17. Li, B., Mellou, K., Zhang, B., Pathuri, J., Menache, I.: Large Language Models for Supply Chain Optimization, (2023). arXiv: 2307.03875 [cs.AI].
18. Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q.V., Zhou, D., Chen, X.: Large Language Models as Optimizers. ArXiv **abs/2309.03409** (2023)
19. Gurobi Optimization, LLC, Gurobi Optimizer Reference Manual, (2024). https://www.gurobi.com.
20. Chen, X., Gao, P., Song, J., Tan, X.: HiQA: A Hierarchical Contextual Augmentation RAG for Multi-Documents QA, (2024). arXiv: 2402.01767 [cs.CL].
21. Gao, L., Ma, X., Lin, J., Callan, J.: Precise Zero-Shot Dense Retrieval without Relevance Labels. In: Rogers, A., Boyd-Graber, J., Okazaki, N. (eds.) Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 1762–1777. Association for Computational Linguistics, Toronto, Canada (2023)

22. Déjean, H., Clinchant, S., Formal, T.: A Thorough Comparison of Cross-Encoders and LLMs for Reranking SPLADE, (2024). arXiv: 2403.10407 [cs.IR].
23. Tam, Z.R., Wu, C.-K., Tsai, Y.-L., Lin, C.-Y., Lee, H.-y., Chen, Y.-N.: Let Me Speak Freely? A Study On The Impact Of Format Restrictions On Large Language Model Performance. In: Dernoncourt, F., Preoţiuc-Pietro, D., Shimorina, A. (eds.) Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, pp. 1218–1236. Association for Computational Linguistics, Miami, Florida, US (2024)
24. Jin, X., Lin, Z.: SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660769
25. Song, Y., Lothritz, C., Tang, X., Bissyandé, T., Klein, J.: Revisiting Code Similarity Evaluation with Abstract Syntax Tree Edit Distance. In: Ku, L.-W., Martins, A., Srikumar, V. (eds.) Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pp. 38–46. Association for Computational Linguistics, Bangkok, Thailand (2024). https://doi.org/10.18653/v1/2024.acl-short.3
26. Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R.J., Javaheripi, M., Kauffmann, P., Lee, J.R., Lee, Y.T., Li, Y., Liu, W., Mendes, C.C.T., Nguyen, A., Price, E., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Wang, X., Ward, R., Wu, Y., Yu, D., Zhang, C., Zhang, Y.: Phi-4 Technical Report, (2024). arXiv: 2412.08905 [cs.CL].
27. DeepSeek-AI, Guo, D., Yang, D., *et al.*: DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, (2025). arXiv: 2501.12948 [cs.CL].
28. Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., Feng, Y., Ren, X., Ren, X., Zhou, J., Lin, J.: Qwen2.5-Coder Technical Report, (2024). arXiv: 2409.12186 [cs.CL].
29. Kalyan, K.S.: A survey of GPT-3 family large language models including ChatGPT and GPT-4. Natural Language Processing Journal **6**, 100048 (2024). https://doi.org/10.1016/j.nlp.2023.100048
30. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. Curran Associates Inc., LA, USA (2022)
31. Schulhoff, S., Ilie, M., Balepur, N., *et al.*: The Prompt Report: A Systematic Survey of Prompt Engineering Techniques, (2025). arXiv: 2406.06608 [cs.CL]. https://arxiv.org/abs/2406.06608.